VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

KNIHOVNA PRO NÁVRH KONVOLUČNÍCH NEURONOVÝCH SÍTÍ

A LIBRARY FOR CONVOLUTIONAL NEURAL NETWORK DESIGN

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PETR REK**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Rek Petr, Bc.**
Obor: Inteligentní systémy
Téma: **Knihovna pro návrh konvolučních neuronových sítí**
A Library for Convolutional Neural Network Design
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou umělých neuronových sítí, zaměřte se na konvoluční neuronové sítě.
2. Navrhněte knihovnu pro efektivní práci s konvolučními neuronovými sítěmi (trénování, inference).
3. Implementujte knihovnu ve zvoleném programovacím jazyce.
4. Ověřte implementaci na zvolených úlohách, které jsou typické pro konvoluční neuronové sítě.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Sekanina Lukáš, prof. Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

V rámci této diplomové práce je čtenář seznámen s problematikou neuronových a konvolučních neuronových sítí. Na základě těchto znalostí je poté proveden návrh a implementace knihovny umožňující práci s konvolučními neuronovými sítěmi – od návrhu, přes trénování až po validaci. Výsledná knihovna je poté vyhodnocena na klasických úlohách pro konvoluční neuronové sítě a porovnána s jinými knihovnami. Rozšířením knihovny, díky kterému se odliší od jiných volně dostupných, je nezávislost na datovém typu. Každá vrstva může mít až tři na sobě nezávislé datové typy – pro váhy, pro inferenci a pro učení. Za účelem vyhodnocení tohoto rozšíření je součástí knihovny i datový typ s pevnou řádovou čárkou. Vliv této reprezentace na přesnost natrénované sítě je podroben experimentům.

Abstract

In this diploma thesis, the reader is introduced to artificial neural networks and convolutional neural networks. Based on that, the design and implementation of a new library for convolutional neural networks is described. The library is then evaluated on widely used datasets and compared to other publicly available libraries. The added benefit of the library, that makes it unique, is its independence on data types. Each layer may contain up to three independent data types – for weights, for inference and for training. For the purpose of evaluating this feature, a data type with fixed point representation is also part of the library. The effects of this representation on trained net accuracy are put to a test.

Klíčová slova

neuronová síť, konvoluční neuronová síť, hluboké učení, klasifikace, regrese, softwarová knihovna, umělá inteligence, učení, pevná řádová čárka, aproximace

Keywords

neural network, convolutional neural network, deep learning, classification, regression, software library, artificial intelligence, learning, fixed point, approximation

Citace

REK, Petr. *Knihovna pro návrh konvolučních neuronových sítí.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Knihovna pro návrh konvolučních neuronových sítí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Rek
18. května 2018

Poděkování

Rád bych poděkoval svému vedoucímu prof. Ing. Lukáši Sekaninovi, Ph.D., za cenné rady při vypracování této diplomové práce.

Obsah

Kapitola 1

Úvod

Konvoluční neuronové sítě spadají do takzvaného *soft computingu*, což jsou výpočetní techniky, které tolerují nepřesnost, nejistotu a částečnou pravdivost v úlohách, které řešíme například v oblasti strojového učení [42]. To je patrné právě u neuronových sítí, které se bez větších zásahů člověka umí naučit řešit problémy, jejichž algoritmické řešení by programátor byl jen velmi obtížně schopen zapsat.

Konvoluční neuronové sítě jsou specializované neuronové sítě, jejichž nejčastější využití spočívá ve zpracování obrazových informací. Což je úkol, na nějž jsou klasické neuronové sítě špatně škálovatelné. Aplikací existuje mnoho, od klasifikace až po detekci objektů v obraze, využívají se například i v autonomních vozidlech nebo také pro zpracování přirozeného jazyka.

V této diplomové práci je popsána problematika konvolučních neuronových sítí a na základě získaných znalostí poté navržena a implementována knihovna realizující celý proces práce s konvolučními neuronovými sítěmi (od návrhu, přes trénování až po jejich využívání – *inferenci*). Čím se knihovna odlišuje od jiných dostupných knihoven, je práce s datovými typy. Síť může zároveň využívat až tři různé typy – pro uložení vah, pro inferenci a pro trénování. Knihovna je také doplněna vlastním datovým typem s pevnou řádovou čárkou.

Text je členěn následovně. V kapitole 2 jsou popsány neuronové sítě, což je v kapitole 3 rozšířeno na konvoluční neuronové sítě. V kapitole 4 je navrženo jádro knihovny, jejíž implementace je popsána v kapitole 5, na což navazuje kapitola 6, kde je na příkladech popsáno, jak knihovnu využít ve vlastním projektu. V kapitole 7 je provedena evaluace knihovny na typických úlohách pro konvoluční neuronové sítě a také provedeno srovnání s dalšími volně dostupnými knihovnami. Je také vyhodnocena úspěšnost implementace typové nezávislosti na sadě experimentů s datovým typem s pevnou řádovou čárkou. V poslední kapitole 8 jsou pak zhodnoceny výsledky této práce.

Aktuálnost práce a rychlost, s jakou se umělá inteligence v posledních letech vyvíjí, dobře shrnuje následující citát:

> „*The pace of progress in artificial intelligence is incredibly fast. Unless you have direct exposure to groups like Deepmind, you have no idea how fast – it is growing at a pace close to exponential.*“
>
> „*Tempo, s jakým se vyvíjí umělá inteligence, je neuvěřitelně rychlé. Pokud se blízce nezajímáte o skupiny jako Deepmind, tak ani nemáte ponětí jak rychle – toto tempo je skoro exponenciální.*“ [1]
>
> Elon Musk [39]

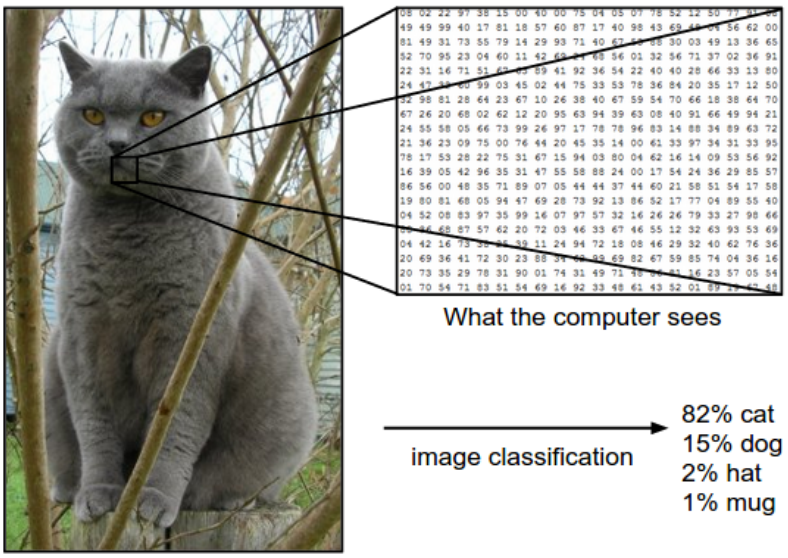
[1] Volně přeloženo autorem.

Kapitola 2

Neuronové sítě

Neuronové sítě jsou vývojovým předstupněm konvolučních neuronových sítí a mají společné principy, které je potřeba znát. Základní neuronová síť je navíc využita jako plně propojená vrstva konvolučních neuronových sítí (popis v této kapitole se tedy soustředí pouze na *plně propojené dopředné neuronové sítě*). V následujících odstavcích budou ve zkratce popsány základní principy neuronových sítí, které poté budou rozvedeny v dalších kapitolách.

Příkladem využití neuronových sítí může být rozpoznání objektu na obrázku. Pro člověka velmi lehký úkol, pro počítač nikoliv. Nejenže v případě počítače se jedná o statisíce až miliony číselných hodnot (viz obrázek 2.1), ale dva obrázky stejné třídy se mohou velmi lišit v závislosti na osvětlení, natočení nebo pozadí (velká *intra-třídní variabilita*).



Obrázek 2.1: Reprezentace obrazových informací v počítači a ukázka možného výsledku neuronové sítě provádějící klasifikaci do čtyř tříd – *kočka, pes, klobouk* a *hrnek* [21]

Přijít s deterministickým algoritmem, který by tuto úlohu zvládal řešit s požadovanou přesností, by bylo velmi obtížné a prakticky jedinou možností je použít univerzální algoritmus, který se problém naučí řešit sám. A právě zde přicházejí ke slovu neuronové sítě, které typicky učíme na příkladech. Tomu říkáme *data-driven approach* (přístup řízený daty) [21]. Začátek využívání neuronových sítí znamenal velký krok kupředu ve vývoji inteligentních systémů.

2.1 Historie

V roce 1943 přišli *Warren S. McCulloch* a *Walter Pitts* s výpočetním modelem založeným na neuronech a jejich propojení (zjednodušení procesů, které se dějí v mozku), který zažehl zájem o neuronové sítě [27].

V roce 1958 *Frank Rosenblatt* oznámil algoritmus nazvaný *Perceptron* [33], což je algoritmus učení s učitelem pro binární klasifikátory. Jedná se o lineární klasifikátor, který využívá lineární funkce, jejímiž vstupy jsou vektor vah (který je předmětem učení) a vektor rysů (vstupní vektor).

Avšak v roce 1969 *Marvin Minsky* a *Seymour Papert* informovali o nedostatcích perceptronu [28]. Přesněji o tom, že perceptron nedokáže napodobit ani tak jednoduchou funkci, jako je *XOR*, a také o tom, že v té době dostupná výpočetní síla není dostatečná pro efektivní trénování neuronových sítí. To na více než 10 let utlumilo zájem o neuronové sítě a jejich výzkum.

Zájem o neuronové sítě opět stoupl v 80. letech 20. století poté, kdy bylo dosaženo významných pokroků v tomto poli (především algoritmus *zpětného šíření chyby* [41] – viz kapitola 2.4.6, který zefektivnil trénování neuronových sítí s více vrstvami a také prakticky vyřešil „*XOR* problém“). Zároveň i výpočetní výkon byl daleko vyšší než dříve.

Od té doby jsou neuronové sítě prakticky neustále ve vývoji a velmi často využívané v mnoha různých odvětvích.

2.2 Biologický neuron

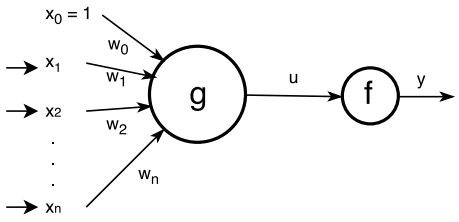
Název *neuronová síť* je odvozen od specializovaných buněk – takzvaných *neuronů*, kterých se v lidském mozku nachází obrovské množství a jsou vzájemně propojeny. Takovému propojení říkáme *synapse*. Každá synapse má svou odpovídající váhu [42].

Zjednodušeně můžeme funkci neuronů popsat následovně. Každý neuron vnímá jiné neurony pomocí *dendritů* a ovlivňuje další neurony pomocí jednoho *axonu*. V případě, že vstupy neuronu přesáhnou jistou hranici, tak neuron vysílá krátký impuls na svém axonu, který periodicky opakuje, dokud vstupy opět neklesnou pod tuto hranici.

Učení neuronů spočívá v nastavování synaptických vah. Umělé neuronové sítě pak do jisté míry napodobují funkci mozku (avšak velmi zjednodušeně).

2.3 Umělý neuron

Práh umělého neuronu nazýváme *bias*, typicky jej modelujeme jako samostatný vstupní neuron, jehož výstup je vždy roven jedné a mění se pouze synaptická váha k němu vedoucí (váha pak představuje zápornou hodnotu prahu).



Obrázek 2.2: Model umělého neuronu

Váhované vstupy neuronu jsou vstupem *bázové funkce* produkující jedinou hodnotu. Typicky používáme *lineární bázovou funkci* (viz rovnice 2.1), méně často pak ve speciálních případech i *radiální bázovou funkci* (viz rovnice 2.2), kde w_i značí váhu vedoucí k i-tému vstupu, x_i značí i-tý vstup a n značí celkový počet vstupů neuronu.

$$u = \sum_{i=0}^{n} w_i x_i \tag{2.1}$$

$$u = \sqrt{\sum_{i=0}^{n} (x_i - w_i)^2} \tag{2.2}$$

Výstup bázové funkce je vstupem *aktivační funkce*, která určuje výstup neuronu. Těchto aktivačních funkcích existuje velké množství a některé z nich budou popsány dále (viz kapitola 2.4.4).

Perceptron Základním modelem umělého neuronu je *perceptron* (viz obrázek 2.2), který používá lineární bázovou funkci g, na jejímž výstupu u je závislý výstup y skokové aktivační funkce f:

$$y = \begin{cases} 1 & \text{pro u} > 0 \\ -1 & \text{pro u} < 0 \\ y_{old} & \text{pro u} = 0 \end{cases} \tag{2.3}$$

Učení vah neuronu pak lze provádět pomocí rovnice 2.5 [42], kde d značí očekávaný výstup, y aktuální výstup, w váhový vektor, x vstupní vektor, η učící koeficient a t vyjadřuje iteraci učení. Rovnice je do jisté míry podobná tomu, jak se učí neuronové sítě, které známe dnes (to bude rozvedeno dále v kapitole 2.4.6).

$$\vec{w}_0 = \text{náhodné} \tag{2.4}$$

$$\vec{w}_t = \vec{w}_{t-1} + \eta \, (d_t - y_t) \, \vec{x_t} \tag{2.5}$$

Jeden takový neuron dokáže klasifikovat lineárně separovatelné prostory – například logický *OR* je řešitelný, avšak *XOR* nikoliv. Řešením je použít více neuronů a uspořádat je do vrstev [26].

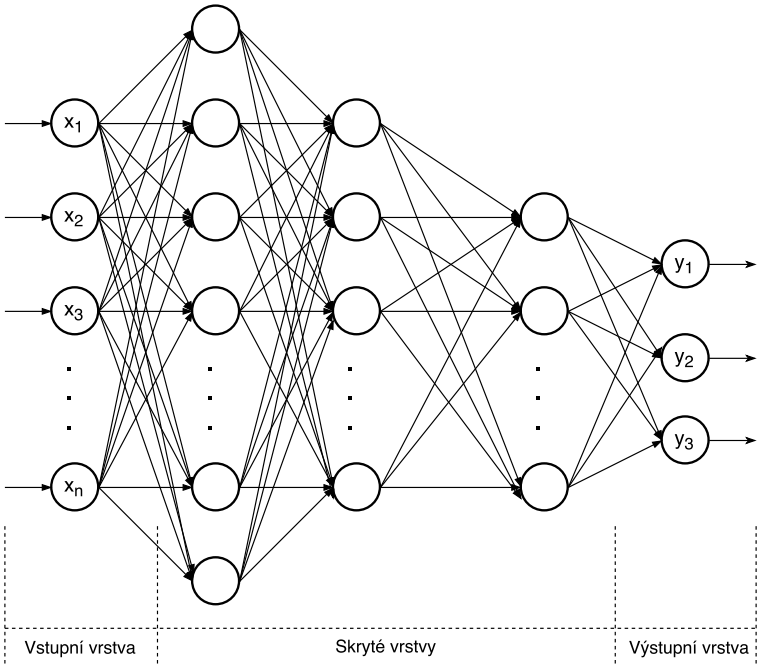
2.4 Umělé neuronové sítě

Neuronová síť je váhovaný orientovaný graf, kde uzly jsou právě umělé neurony. Architektur neuronových sítí existuje větší množství, my se v této kapitole omezíme pouze na neuronové sítě, které jsou využívány v konvoluční neuronové síti jako plně propojená vrstva.

V našem případě se tedy jedná o acyklický váhovaný orientovaný graf, kde uzly jsou organizovány do vrstev (každá vrstva může mít jiný počet neuronů). Spojovat lze pouze neurony v sousedních vrstvách. Tuto architekturu neuronové sítě nazýváme *dopředná neuronová síť*. Neuron je také dále propojen se všemi neurony v předchozí vrstvě – tuto architekturu pak nazýváme *plně propojená dopředná neuronová síť*.

První vrstva obsahuje vstupní hodnoty a říká se jí tedy *vstupní vrstva*. Výstupy poslední vrstvy jsou výstupy celé neuronové sítě a říkáme jí tedy *výstupní vrstva*. Zbylé vrstvy nazýváme *skryté vrstvy* (nemusí být žádná, ale mohou jich být i desítky – to už vede na

takzvaný *deep learning* – hluboké učení). Počet neuronů vstupní a výstupní vrstvy je dán aplikací, počet skrytých vrstev (a neuronů v nich) je volen programátorem v závislosti na složitosti řešené úlohy. Neexistuje jednoznačné doporučení kolik skrytých vrstev a jaký počet neuronů v nich zvolit.



Obrázek 2.3: Příklad umělé neuronové sítě se třemi skrytými vrstvami

Výhodou neuronových sítí je jednak jejich univerzalita (dokáží řešit i úlohy, které by byly velmi obtížně řešitelné klasickými algoritmy), ale i rychlost (i když trénování může trvat velmi dlouhou dobu, tak následné využívání natrénované sítě je časově relativně nenáročné). Problémem ale může být prostorová náročnost, neboť velké neuronové sítě mají i desítky milionů vah, které je třeba uložit (viz kapitola 2.5).

2.4.1 Trénování a validace

Neuronové sítě mají schopnost učení dánu postupnými úpravami synaptických vah. Jako vstup při trénování dostávají množinu dvojic $\{(x_0, y_0), (x_1, y_1), ...(x_n, y_n)\}$, kde x_i je vstup a y_i očekávaný výstup. Neurony pak postupně upravují váhy tak, aby s každou iterací byl výstup sítě blíže očekávané hodnotě než v předchozí iteraci. To nazýváme *konvergencí* neuronové sítě.

Přístupů k učení vah existuje větší množství, avšak algoritmus *zpětného šíření chyby* (anglicky *backpropagation*) dominuje. Algoritmus bude popsán dále v kapitole 2.4.6 poté, co budou vysvětleny další podstatné pojmy.

Vyhodnocení úspěšnosti sítě se typicky neprovádí na trénovací sadě (vzhledem k možnosti přeučení, což je problém zmíněný dále v kapitole 2.4.2), ale slouží k tomu testovací sada, průnik obou sad je prázdný. Ta je typicky mnohem menší než trénovací sada – časté rozdělení je 80 : 20 nebo 90 : 10 *(trénovací* : *testovací)*. V případě klasifikace například zjišťujeme poměr správně klasifikovaných obrázků testovací sady vůči počtu dvojic celé sady.

2.4.2 Nevýhody neuronových sítí

I když jsou neuronové sítě velmi často využívány vzhledem k jejich mnoha přednostem, existuje i několik problémů, na které je třeba dát si pozor. Ty budou v této části zmíněny.

Citlivost na parametry Neuronové sítě jsou při učení velmi citlivé na zvolené parametry učení. Špatné nastavení parametrů může způsobit, že síť nebude konvergovat ke správnému řešení. Příkladem takového parametru může být učící koeficient η. V případě příliš malé hodnoty trvá učení dlouho, ale prakticky vždy konverguje. V případě vysoké hodnoty se síť učí rychleji, ale může snadno dojít k divergenci.

Doba učení V závislosti na zvolené úloze (a s tím související velikostí sítě) může učení neuronové sítě trvat velmi dlouhou dobu. To je zejména patrné u velmi velkých vstupních vektorů a sítí s mnoha skrytými vrstvami a velkým počtem neuronů v nich.

Schopnost najít řešení Ne vždy je síť schopna najít řešení – to souvisí s její schopností učit se. Pokud existuje velké množství vstupních proměnných, není síť složená z malého počtu neuronů schopna je všechny „pokrýt“ a nedokáže překonat určitou hranici úspěšnosti [1]. Tomu říkáme, že síť není schopna realizovat požadovanou funkci. Řešením je zvětšení počtu neuronů a skrytých vrstev, avšak to nás pak může přivést k problému zvanému *overfitting* popsanému dále.

Obdobně v případě, že síti poskytneme příliš velké množství parametrů (více než je nutno, tedy některé z nich jsou redundantní), tak se síť sice bude schopná učit, avšak rychlost konvergence bude nižší, než by tomu bylo v případě, že by vstupní vektor redundantní parametry neobsahoval. Často je tedy vhodné provést předzpracování dat.

Přeučení Přeučení, neboli *overfitting*, nastává tehdy, pokud se síť ve stejnou chvíli zlepšuje na trénovacích datech, ale zároveň zhoršuje na datech testovacích. Existují dva hlavní důvody, proč k přeučení může dojít:

- v případě, že neuronová síť obsahuje příliš mnoho neuronů a vrstev, tak je schopna se přesně naučit trénovací vstupy,
- v případě, že síť učíme velmi dlouhou dobu, je možné, že se postupem času spíše přizpůsobí konkrétním vstupům.

V obou případech síť ztrácí schopnost zobecňovat (*generalizace*) a vede si hůře na testovacích datech, která nikdy neměla na vstupu.

Neinterpretovatelnost Jedním z problémů je také neinterpretovatelnost naučené sítě. Z naučených vah je velmi obtížně možno poznat, jak síť dochází k učiněným závěrům. To znesnadňuje nasazení řešení založených na neuronových sítích v procesech, jejichž selhání by mohlo mít fatální následky. Stejně tak není lehké odhalit slabiny sítě za účelem jejího „doučení“.

[1] To je dobře patrné v demonstrační aplikaci dostupné zde: http://playground.tensorflow.org/.

2.4.3 Typy úloh

Rozlišujeme několik přístupů k učení neuronových sítí [18], které jsou blíže popsány v následujícím seznamu:

- posilované učení – trénování je prováděno na množině příkladů sestávajících se pouze ze vstupů, není dána žádná informace o očekávaném výstupu, avšak během učení je algoritmus odměňován na základě kvality aktuálního řešení,
- učení bez učitele – trénování je opět prováděno na množině příkladů sestávajících se pouze ze vstupů a algoritmus nemá k dispozici žádné informace o očekávaných výstupech nebo kvalitě aktuálního řešení (příkladem využití je shlukování),
- učení s učitelem – u neuronových sítí je nejčastější, trénování je prováděno na množině příkladů skládajících se z dvojice vstupu a očekávaného výstupu, na základě toho lze spočítat chybu neuronové sítě a využít ji k úpravě vah.

V případě učení s učitelem, které nás bude zajímat v této práci, rozlišujeme dva hlavní typy úloh, které budou v následujících odstavcích představeny.

Klasifikace Úlohou klasifikace je pro vstupní vektor určit třídu, nebo více tříd, do které spadá. Počet tříd je předem znám a trénovací data neuronové sítě jsou pak dvojicemi *(vstupní vektor, očekávaná třída)*. Každý výstupní neuron pak reprezentuje jednu ze tříd (třídě je přiřazena číselná hodnota dle pořadí hodnoty výstupního neuronu ve výstupním vektoru). Velmi často je také žádoucí, aby výstupní hodnoty udávaly pravděpodobnost příslušnosti vstupního vektoru do dané třídy.

Regrese Cílem regresní úlohy je pro vstupní vektor předpovědět hodnoty výstupního vektoru. Názorným příkladem může být aproximace funkce, kdy vstupní vektor představuje hodnoty proměnných a výstupní vektor pak výsledek aproximované funkce (výstupních hodnot je ale typicky více). Trénovací data jsou formátu *(vstupní vektor, očekávaný výstupní vektor)*.

2.4.4 Aktivační funkce

Na rozdíl od bázové funkce, která nás v našem případě zajímá pouze jedna (*lineární bázová funkce*), tak aktivačních funkcí budeme uvažovat více. Některé z nich jsou zmíněny níže. Velmi jednoduchou skokovou aktivační funkci bylo možno již vidět v rovnici 2.3.

Sigmoida Typická aktivační funkce, která je využívána téměř v každé knize o neuronových sítích jako příklad, je sigmoida. Její výstup leží v intervalu $(0, 1)$ a je dán následujícím vzorcem:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

Používala se zejména dříve, avšak nyní ztratila na popularitě kvůli různým nedostatkům [21], a to:

- v případě, že vstup nebo výstup je velmi vysoký (respektive nízký), je následně gradient velmi nízký, což má neblahý vliv na rychlost učení,

- počáteční váhy neuronů musí být nastaveny opatrně, aby nedošlo k saturaci (výstup se blíží 0 nebo 1), pak se neuron velmi pomalu učí (viz předchozí bod),
- výstupní hodnota nemá střed v nule, což má neblahý vliv na další vrstvy a může způsobovat nežádoucí výkyvy v učení.

Tanh (Hyperbolický tangens) Výstup funkce náleží do intervalu $(-1, 1)$. Je velmi podobný sigmoidě, ale na rozdíl od ní mají výstupní hodnoty střed v bodě nula.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.7}$$

ReLU Zkratka náleží anglickému *Rectified linear unit*. Funkce je velmi jednoduchá, ale současně populární v posledních letech. Experimentálně navíc bylo zjištěno, že konverguje mnohem rychleji než dříve zmíněné aktivační funkce [9].

$$f(x) = \begin{cases} x & \text{pro x} \geq 0 \\ 0 & \text{pro x} < 0 \end{cases} \tag{2.8}$$

Problémem *ReLU* aktivační funkce je však možnost *umrtvení* [21], kdy může dojít k tomu, že neuron v budoucnu již nikdy nebude aktivován a gradient proudící skrze neuron bude vždy nula (nedojde tedy k nápravě). Tento problém je patrný zejména při vyšším učícím koeficientu η a snaží se jej řešit aktivační funkce *Leaky ReLU*, viz dále.

Leaku ReLU Aktivační funkce *Leaky ReLU* v případě, že vstup je menší jak nula, tento vstup vynásobí malou konstantou, např. 0.01, jak je ukázáno v rovnici níže. Právě proto, že i pro vstupy menší než nula nebude gradient nulový, nedojde k umrtvení neuronu (respektive neuron má šanci se zotavit).

$$f(x) = \begin{cases} x & \text{pro x} \geq 0 \\ 0.01x & \text{pro x} < 0 \end{cases} \tag{2.9}$$

Softmax Aktivační funkce *Softmax* je velmi často používána u klasifikačních úloh, kdy je často žádoucí, aby výstupy náležely do intervalu $\langle 0, 1 \rangle$ a udávaly pravděpodobnost toho, že vstup patří do třídy dané výstupním neuronem. Právě takovou úpravu provádí aktivační funkce *Softmax*, která se typicky používá ve výstupní vrstvě neuronové sítě provádějící klasifikaci. Je dána následujícím vzorcem (k výpočtu je potřeba hodnot všech K výstupních neuronů):

$$f(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \tag{2.10}$$

2.4.5 Ztrátové funkce

Ztrátové funkce, v angličtině nazývané *loss functions* nebo také *cost functions*, vyjadřují aktuální chybu sítě během trénování. Jedná se o hodnotu, která vyjadřuje rozdíl mezi aktuálním výstupem a očekávaným výstupem. Výstup této funkce je poté základem pro učící algoritmus, který se chybu postupně snaží minimalizovat.

Volba vhodné ztrátové funkce může mít velký vliv na rychlost a kvalitu učení neuronové sítě. Dvě velmi často používané chybové funkce jsou zmíněny níže [13], existuje však řada dalších.

Střední kvadratická chyba Anglicky *Mean squared error*, také známa jako *Quadratic cost*, se používá velmi často pro regresní úlohy a je definována následovně:

$$E(\vec{o}, \vec{t}) = \frac{1}{n} \sum_{i=1}^{n} (t_i - o_i)^2 \,, \tag{2.11}$$

kde t je očekávaný výstup, o aktuální výstup a n je počet výstupních neuronů.

Cross-entropy pro binární klasifikaci Tento typ ztrátová funkce *cross-entropy* je využíván při klasifikaci do dvou tříd (stačí jeden výstupní neuron, jehož očekávaný výstup je například buď 1, nebo 0). Zápis funkce lze vidět v rovnici 2.12, kde t je očekávaný výstup, o aktuální výstup a n je počet výstupních neuronů. Suma je v rovnici z toho důvodu, že uvažujeme i několik na sobě nezávislých binárních klasifikací (každá taková klasifikace má tedy vlastní výstupní neuron).

$$E(\vec{o}, \vec{t}) = \sum_{i=1}^{n} -t_i \, log(o_i) - (1 - t_i) \, log(1 - o_i) \tag{2.12}$$

Cross-entropy pro klasifikaci do více tříd Tento typ ztrátové funkce *cross-entropy* je využíván při klasifikaci do tří a více tříd (kdy každé třídě odpovídá jeden výstupní neuron). Funkce je definována následovně:

$$E(\vec{o}, \vec{t}) = \sum_{i=1}^{n} -t_i \, log(o_i) \tag{2.13}$$

2.4.6 Algoritmus zpětného šíření chyby

Algoritmus zpětného šíření chyby, známý zejména pod anglickým názvem *backpropagation*, patří pod metody souhrnně nazývané *supervised learning*, tedy učení s učitelem (neboť je učen na příkladech, ke kterým je známa správná „odpověď“). V současnosti se jedná o nejvíce využívaný algoritmus pro učení neuronových i konvolučních neuronových sítí. Učení probíhá v iteracích, které nazýváme *epochy*.

Učení s učitelem lze formálně popsat tak, že hledáme funkci h (nazýváme *hypotézou*), která je co nejvíce podobná cílové funkci f ($h \approx f$) a to trénováním na dvojicích $(x, f(x))$ [19]. Přístupů k učení s učitelem existuje více – v našem případě se jedná o takzvané *konekcionistické učení*.

Učení je v případě algoritmu *backpropagation* založeno na hledání minima ztrátové funkce pomocí *gradientního sestupu* (anglicky *gradient descent*). Vzhledem k nutnosti výpočtu gradientů je nutné, aby ztrátová funkce byla derivovatelná (derivovatelné musí být i aktivační funkce a další operace prováděné při výpočtu – to je podstatné zejména později u konvolučních neuronových sítí).

Gradientní sestup počítá gradient ztrátové funkce vzhledem k jednotlivým vahám a pohybuje se v opačném směru, aby minimalizoval chybu sítě na trénovací sadě.

Názorný příklad

Výpočet gradientu si představíme na jednoduchém příkladu [19] jednoho neuronu, pro nějž ztrátovou funkci definujeme jako:

$$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2, \tag{2.14}$$

kde D je množina trénovacích příkladů, t očekávaný výstup a o výstup neuronu.

Rovnice je podobná ztrátové funkci *střední kvadratické chyby*, avšak zahrnuje všechny vzory trénovací sady. Dělení dvěma na začátku se často používá vzhledem k tomu, že při derivaci funkce dojde ke zjednodušení (vzhledem k umocnění dvěma).

Výpočet gradientu ztrátové funkce je pak postupně rozepsán v rovnicích 2.15 a 2.18.

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 = \\ &= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) = \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) = \\ &= \sum_{d\in D}(t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) = -\sum_{d\in D}(t_d - o_d)\frac{\partial o_d}{\partial(\vec{w}\vec{x_d})}\frac{\partial(\vec{w}\vec{x_d})}{\partial w_i}\end{aligned} \tag{2.15}$$

$$\frac{\partial(\vec{w}\vec{x_d})}{\partial w_i} = x_{i,d} \tag{2.16}$$

Pokud dále budeme uvažovat například aktivační funkci *sigmoida*, kterou označíme σ, pak lze v rovnici 2.17 vyjádřit i zbylou derivaci. S uvažováním rovnic 2.16 a 2.17 pak můžeme odvodit výslednou rovnici udávající gradient ztrátové funkce vzhledem k jednotlivým vahám neuronu, viz rovnice 2.18.

$$\frac{\partial o_d}{\partial(\vec{w}\vec{x_d})} = \frac{\partial\sigma(\vec{w}\vec{x_d})}{\partial(\vec{w}\vec{x_d})} = o_d(1 - o_d) \tag{2.17}$$

$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D}(t_d - o_d)o_d(1 - o_d)x_{i,d} \tag{2.18}$$

Na základě toho pak můžeme definovat rovnici 2.19, která je používána pro úpravu vah neuronu. Při úpravě se tedy pohybujeme v opačném směru ke gradientu ztrátové funkce, a to s krokem daným konstantou – učícím koeficientem η. Jedná se o ten nejzákladnější algoritmus úpravy vah, další budou představeny v kapitole 2.4.7.

$$w_i = w_i - \eta\frac{\partial E}{\partial w_i} \tag{2.19}$$

Algoritmus zpětného šíření chyby (backpropagation)

Rovnici 2.18 lze použít jen v případě neuronové sítě bez skrytých vrstev, což je velmi omezující. Při trénování neuronových sítí s více vrstvami je nutno vypočtenou chybu, jak název algoritmu napovídá, propagovat na předcházející vrstvy.

Na každý neuron tedy propagujeme chyby, na kterých se podílel, a to s váhou, s jakou se na nich podílel. Celá myšlenka zpětného šíření chyby je znázorněna v algoritmu 1, kde je chyba již uvažována pro jednotlivé trénovací vzorky (tedy bez sumy, jedná se o stochastický gradientní sestup – viz kapitola 2.4.7). Využívána je opět aktivační funkce *sigmoida* a střední kvadratická odchylka jako ztrátová funkce.

Algoritmus 1: Zpětné šíření chyby

inicializuj všechny váhy $w_{i,j}$ *na náhodnou nízkou hodnotu*
while *není výsledek dostatečný* **do**
  for *každý trénovací vzorek* **do**
    spočítej výstupy $\vec{o}$ *každé vrstvy neuronové sítě pro vstup vrstvy* $\vec{x}$
    for *každý výstupní neuron* k **do**

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

    end for
    for *každou skrytou vrstvu od konce* **do**
      for *každý neuron* h *v této vrstvě* **do**

$$\delta_h = o_h(1 - o_h) \sum_{k \in \text{následující vrstva}} w_{h,k}\delta_k$$

      end for
    end for
    for *každou váhu* $w_{i,j}$ **do**

$$w_{i,j} = w_{i,j} + \eta\delta_j x_{i,j}$$

    end for
  end for
end while

Tento algoritmus sebou samozřejmě nese i různé problémy, například:

- může dojít k uváznutí v lokálním minimu (a tedy není možno nalézt nejlepší řešení),
- může nastat předčasné ukončení učení kvůli nulovému gradientu (je tedy vhodné používat aktivační funkce, jejichž gradient není nikdy nulový – ale může být velmi malý).

Existují samozřejmě i další přístupy k učení neuronových sítí, avšak žádný z nich zatím nepřekonal výhody *backpropagation* a příliš se nepoužívá. Jako příklad můžeme zmínit vy-

užití *genetických algoritmů* pro učení vah [14, 32]. I když se jedná o zajímavý přístup, tak rychlost konvergence je velmi nízká vzhledem k obrovské velikosti prohledávaného prostoru.

2.4.7 Gradientní sestup

Gradientní sestup (anglicky *gradient descent*) se používá pro optimalizaci neuronových a také konvolučních neuronových sítí. Cílem gradientního sestupu je minimalizovat zvolenou ztrátovou funkci a to úpravou učitelných parametrů sítě založenou na pohybování se v opačném směru k jejímu gradientu (viz výše). Základní variantu bylo možno vidět v rovnici 2.19.

Základním parametrem je η, jedná se o učící koeficient, který ovlivňuje velikosti kroků, které provádíme za účelem dosažení (lokálního) minima dané funkce. Jeho hodnota je velmi podstatná – v případě malých hodnot trvá trénování velmi dlouho, avšak v případě velkých hodnot může síť špatně konvergovat nebo dokonce divergovat. Někdy se také používá přístup postupného snižování učícího koeficientu se stoupajícím počtem epoch trénování.

Gradientní sestup lze podle frekvence úpravy učitelných parametrů rozdělit na tři základní kategorie.

- Dávkový gradientní sestup (anglicky *batch gradient descent*) – úprava vah se provádí po průchodu celou trénovací množinou. Konvergence je pomalá, ale při vhodně zvoleném učícím koeficientu zaručená.
- Stochastický gradientní sestup (anglicky *stochastic gradient descent*) – úprava vah se provádí pro každý vzorek okamžitě. Konvergence je rychlejší, avšak může docházet k fluktuacím.
- Gradientní sestup s menšími dávkami (anglicky *minibatch gradient descent*) – kombinuje výhody obou předchozích, jedna dávka má velikost n (platí $1 < n < m$, kde m je celkový počet příkladů trénovací sady). Používají se různé velikosti n, často se jedná o mocniny dvou v rozmezí $2 - 256$.

To si lze jednoduše ilustrovat na algoritmu 2. Jednotlivé metody se pak liší podle implementace řádku 3 – v případě dávkového přístupu získáme celou trénovací množinu, v případě stochastického přístupu získáme jeden vzorek a v případě menších dávek (*minibatch*) pak dávku zvolené velikosti.

Algoritmus 2: Úprava vah neuronové sítě

```
1:  while není výsledek dostatečný do
2:      while nebyly použity všechny trénovací vzorky do
3:          získej aktuální dávku vzorků
4:          for každý vzorek v dávce do
5:              spočítej gradient pro každý parametr
6:              akumuluj gradient pro každý parametr
7:          end for
8:          použij průměr akumulovaných gradientů k úpravě každého z parametrů
9:      end while
10: end while
```

Velmi častým rozšířením je „zamíchání“ trénovacích vzorků před každou epochou, což má typicky příznivý vliv na kvalitu učení (s výjimkou *dávkového gradientního sestupu*, kde to nehraje roli).

Optimalizační metody

Existuje řada variant [34, 21] gradientního sestupu, jejichž cílem je kvalitnější učení, ale také snížení závislosti na vhodné volbě parametrů. Tyto metody se souhrnně označují jako optimalizátory. V této kapitole budou shrnuty některé často používané.

V odstavcích níže budou použita následující značení:

- θ – parametr, který je upravován,
- $\nabla E(\theta)$ – gradient ztrátové funkce vzhledem k parametru θ.

Klasickou metodu gradientního sestupu pak lze v tomto případě zapsat jako:

$$\theta_t = \theta_{t-1} - \eta \nabla E(\theta_{t-1}) \tag{2.20}$$

Momentum Kromě učícího koeficientu η zavádí parametr momenta γ. Typickou hodnotou je 0.9 [34]. Rovnice pro úpravu parametrů pak vypadá následovně:

$$v_t = \gamma v_{t-1} + \eta \nabla E(\theta_{t-1}) \tag{2.21}$$

$$\theta_t = \theta_{t-1} - v_t \tag{2.22}$$

V důsledku momentum znamená, že posilujeme po sobě jdoucí úpravy „mířící“ stejným směrem a penalizujeme úpravy mířící jiným směrem. To zrychluje konvergenci a omezuje fluktuaci.

Nestorovo momentum Často používanou variantou momenta je využití Nestorova momenta, které v mnoha úlohách vede ke značnému zlepšení konvergence parametrů. Získáváme tak schopnost uvažovat i budoucí polohu parametrů, což umožní lepší responzivitu.

$$v_t = \gamma v_{t-1} + \eta \nabla E(\theta_{t-1}) \tag{2.23}$$

$$\theta_t = \theta_{t-1} - ((-\gamma)\, v_{t-1} + (1+\gamma)\, v_t) \tag{2.24}$$

Nestorovo momentum nám umožňuje adaptovat úpravy vah vzhledem k tvaru chybové funkce. Dalším krokem je pak adaptování úprav vah vzhledem ke konkrétním parametrům a jejich důležitosti – k čemuž slouží dále zmíněný optimalizátor *Adagrad*.

Adagrad Adagrad je optimalizátorem, který dynamicky upravuje koeficient učení vzhledem k jednotlivým parametrům. Provádí větší změny pro méně časté parametry a menší změny pro častější parametry. Tím tedy odpadá starost o postupné změny učícího koeficientu uživatelem. ϵ je nízká hodnota (typicky 10^{-8}), pomocí které se vyhýbáme dělení nulou. Nevýhodou však je, že akumulovaná hodnota G_t (viz rovnice 2.25) se postupem času zvětšuje, což způsobuje zmenšování velikosti učiněných kroků, a to až do fáze, kdy se učení téměř neprovádí.

$$G_t = G_{t-1} + \nabla E(\theta_{t-1})^2 \tag{2.25}$$

$$\theta_t = \theta_{t-1} - \frac{\eta \nabla E(\theta_{t-1})}{\sqrt{G_t} + \epsilon} \tag{2.26}$$

Adam Zkratka pro anglické *ADAptive Moment estimation* je také metodou, která upravuje učící koeficient pro každý parametr zvlášť. Tento optimalizátor často dosahuje nejlepších výsledků ze všech zmíněných dříve.

Doporučené hodnoty parametrů jsou $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ [34].

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla E(\theta_{t-1}) \tag{2.27}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla E(\theta_{t-1})^2 \tag{2.28}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.29}$$

$$\theta_t = \theta_{t-1} - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.30}$$

Regularizace Velmi často se každá z optimalizačních metod doplňuje parametrem μ provádějícím periodické snižování vah (anglicky *weight decay*). Úprava vah je poté dále doplněna vzorcem:

$$\theta_t = \theta_{t-1} - \eta\, \mu\, \theta_{t-1} \tag{2.31}$$

V závislosti na implementaci se v rovnici 2.31 někdy η (učící koeficient) nemusí vyskytovat, často je ale vhodnější používat μ relativní k η, tak jak je ukázáno zde.

Tento parametr do jisté míry zabraňuje tomu, aby se váhy staly příliš velkými. To má neblahý vliv na učení a často je také příznakem přeučení sítě.

2.4.8 Nastavení trénování

Aby síť konvergovala (a to rychle) ke správnému nastavení vah, je nutno zvolit vhodné počáteční parametry neuronové sítě a trénování. V souhrnu níže budou uvedeny některé podstatné principy [42].

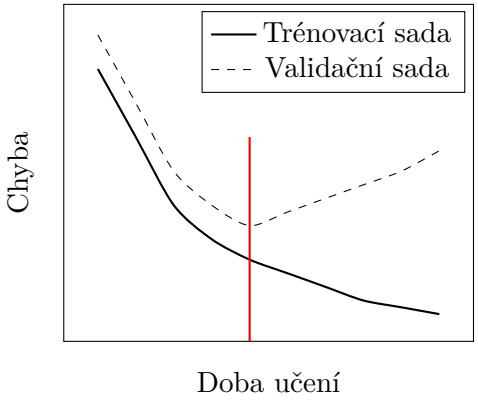
Počáteční nastavení vah Váhy před trénováním by měly být nastaveny na nízkou hodnotu v okolí nuly. V zásadě platí, že čím více vstupů neuronu, tím menší by počáteční váhy měly být, aby ihned po spuštění nedošlo k přetečení (jako v případě aktivační funkce $ReLU$) nebo saturaci (jako v případě aktivační funkce *sigmoida*). O tom, jak vhodně nastavit váhy, lze nalézt velké množství studií. O žádném nastavení nelze prohlásit, že by bylo univerzální. Často se však používá nastavení na náhodnou hodnotu z intervalu $\langle -\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}} \rangle$ a obdobné varianty, kde n je počet vstupů neuronu.

Koeficient učení Koeficient učení ovlivňuje rychlost konvergence vah neuronové sítě. V případě nízké hodnoty se síť učí pomalu, v případě vysoké hodnoty však může dojít k divergenci. Většinou se volí hodnota z intervalu $(0, 1\rangle$, která je opět silně závislá na zvolené aktivační a ztrátové funkci. Typicky se ale jedná o velmi malou hodnotu (například 0.01).

Alternativou je dynamické nastavování koeficientu učení a to tak, že na počátku je hodnota vyšší a postupně se snižuje. Tento přístup zajistí rychlejší konvergenci na počátku a zároveň zabrání divergenci v pozdějších fázích učení.

Výběr trénovacích vzorků Při učení lze procházet trénovací sadu sekvenčně, vzorek po vzorku. Existuje ale i přístup náhodného vybírání vzorků (respektive zamíchání trénovací sady před začátkem každé epochy). To může vést na zlepšení výsledků.

Ukončení učení Již byl zmíněn problém přeučení sítí, který nazýváme *overfitting*, kdy síť sice dosahuje čím dál lepších výsledků na trénovací sadě, ale zároveň klesá její úspěšnost na dosud neznámých vstupech. Je proto nutné ukončit učení dříve, než k tomu dojde. Často se používá ukončení učení v případě, že chyba klesne pod zadanou hodnotu. Případně se po každé epoše provádí testování sítě nad validační sadou a trénování se ukončí v momentě, kdy začne klesat úspěšnost (nebo se přestane zlepšovat) – v angličtině toto řešení nazýváme *early stopping* (brzké zastavení). Tento problém lze vidět na obrázku 2.4.



Obrázek 2.4: Znázornění přeučení sítě, svislá červená čára znázorňuje ideální konec učení

2.5 Přechod ke konvolučním neuronovým sítím

Moderní neuronové sítě obsahují i desítky vrstev. Relativně malý černobílý obrázek 250×250 pixelů vyústí v 62 500 neuronů ve vstupní vrstvě. V případě, že následující vrstva bude mít stejný počet neuronů, tak se dostáváme na 16 000 000 parametrů, což dále můžeme vynásobit počtem vrstev (i když počet neuronů ve vrstvách se typicky postupně snižuje). To znamená velmi vysoké paměťové, ale i časové nároky. Paměťové se projeví jak při využívání, tak při trénování. Časové pak zejména při trénování.

Prvním podstatným rozdílem konvolučních neuronových sítí je, že dimenzionalita vstupu je uchovávána po celou dobu průchodu sítí až po plně propojenou vrstvu na jejím konci. Pracujeme tedy se šířkou, výškou i hloubkou (např. počet barevných kanálů) oproti jednorozměrnému vstupnímu vektoru v případě neuronových sítí.

Druhým, podstatnějším rozdílem, je způsob výpočtu. Konvoluční neuronové sítě mají svůj název podle konvolučních vrstev. Ty neoperují na celém vstupu najednou, ale po menších kusech. To velmi významně redukuje počet učených parametrů. Tento typ výpočtu nazýváme *lokální konektivitou*.

Konvoluční neuronové sítě mají s neuronovými sítěmi mnoho společného a prakticky všechny znalosti zmíněné v této kapitole lze také aplikovat. Konvoluční neuronové sítě také obsahují klasické neuronové sítě jako jednu z vrstev.

Kapitola 3

Konvoluční neuronové sítě

Konvoluční neuronové sítě (dále i CNN, podle anglického *Convolutional Neural Network*) jsou zvláštním případem klasických neuronových sítí, které slouží pro zpracování dat vykazujících určitou strukturovanost [15]. V praxi se typicky ztotožňují s počítačovým viděním, využívají se například v autonomních vozidlech. Využití ale existuje více, poslední dobou se čím dál více využívají i pro zpracování přirozeného jazyka [5].

Typickým příkladem strukturovanosti je černobílý obrázek, který má 2D (3D pro barevný) strukturu a dává smysl, pouze pokud je tato struktura zachována. V případě, že dojde k přehození sloupců či řádků pixelů, obrázek ztrácí svůj původní význam. Tato uspořádanost platí i pro zvuk.

Název *konvoluční* pochází z matematické operace *konvoluce*, která bude popsána v kapitole 3.2. Někdy je můžeme také nalézt pod názvem *shift invariant artificial neural networks* (volně lze přeložit jako *umělé neuronové sítě nezávislé na posuvu*) právě kvůli konvoluční vrstvě, kde váhy filtrů jsou nezávislé na poloze ve vstupu.

3.1 Historie

Konvoluční neuronové sítě se inspirovaly ve zpracování vizuálních informací živými organismy, kdy v oku existují *receptory*, které do jisté míry byly předlohou filtrů, o kterých se budeme bavit dále.

Jako začátek praktického využívání konvolučních neuronových sítí můžeme označit 90. léta spojená se sítěmi *LeNet*, zejména architekturou s názvem *LeNet-5* [24], kterou představil *Yann LeCun*. Používala se na rozpoznávání číslic v obraze.

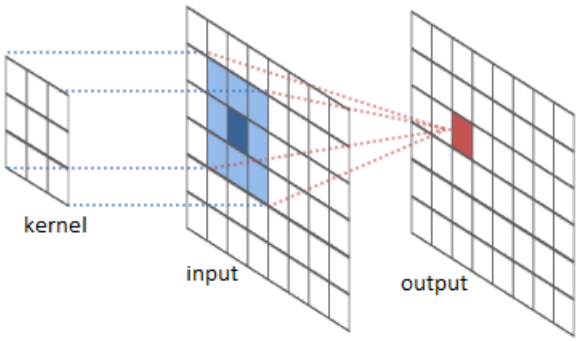
Velký skok kupředu nastal v roce 2012, kdy *Alex Krizhevsky*, *Ilya Sutskever* a *Geoff Hinton* přišli s konvoluční neuronovou sítí obsahující více než tisíc filtrů v pěti konvolučních vrstvách, *ReLU* aktivační funkci a *Dropout* vrstvou (řešící overfitting). Ta navíc byla trénována na GPU a to na velkém množství obrázků z 1 000 tříd [23]. Na výsledky této práce pak další navázali. Další známé architektury lze nalézt v kapitole 3.5.1.

3.2 Konvoluční operace

Jádrem konvolučních neuronových sítí jsou *konvoluční vrstvy*, které, jak z názvu vyplývá, využívají operace *konvoluce*. Během konvoluce by dle definice mělo před posouváním filtru po vstupu dojít k přetočení filtru, v konvolučních neuronových sítí se však často od tohoto

kroku upouští – v tom případě se operace správně nazývá *cross-correlation*, ale stále se používá označení „konvoluce“ [15, 40]. Stejně tomu bude i v této práci.

Nejjednodušeji si tedy můžeme prováděnou operaci v CNN představit jako posouvání okna po vstupní matici a násobení překrývajících se hodnot. Tomuto oknu říkáme *filtr* (často ale také *jádro*, *kernel* nebo *feature detector*) a je typicky mnohem menší než vstupní obrázek. Pro hodnoty uvnitř filtru existuje více výrazů, v této práci je budeme nazývat váhami – paralela ke klasickým neuronovým sítím. Každé posunutí okna produkuje jednu výstupní hodnotu, ze kterých je poté složena výstupní matice, kterou nazýváme *feature map*. Čím více filtrů konvoluční vrstva má, tím více rysů se naučí detekovat.



Obrázek 3.1: Vizualizace konvoluce [38]

Tento postup byl znám daleko dříve, než vznikly CNN. Filtry se již dlouhou dobu používají při zpracování obrazu – rozmazání, detekce hran atd. V rámci naučené konvoluční neuronové sítě si pak často můžeme povšimnout toho, že podobné filtry (co do jejich vah) se síť sama naučí [23].

Co se týče analogie k neuronovým sítím, tak si můžeme představit konvoluční vrstvy jako neuronové sítě, kde se jeden neuron vyskytuje vícenásobně se stejnými váhami, ale je vždy připojen k jiné kombinaci vstupních neuronů (tento neuron odpovídá jednomu filtru). Zatímco počet operací se tedy nesníží, tak počet parametrů, které je třeba se naučit a uložit, se rapidně zmenší.

Oproti neuronovým sítím se zde klade důraz na hledání lokální informace, nedíváme se na vstup jako na celek, ale jako na množinu jeho částí. Konvoluční vrstvy se pak vrství za sebe, a zatímco filtry v počátečních vrstvách vyhledávají prosté útvary – například horizontální a vertikální „hrany“ v obličeji, vrstvy na konci detekují složitější tvary – například oči nebo ústa.

3.3 Trénování

Trénování konvoluční neuronové sítě je opět založeno na algoritmu *backpropagation*, který byl popsán u neuronových sítí. V rámci plně propojené vrstvy je nezměněn, v dalších vrstvách se pak již liší. Tyto změny budou popsány v další kapitole u jednotlivých vrstev.

Z globálního hlediska učení probíhá následovně. Na základě ztrátové funkce je vypočtena chyba jednotlivých výstupních neuronů (podle rozdílu aktuálního a očekávaného výstupu). Předpokládejme například, že poslední vrstvou je plně propojená vrstva. Ta obdrží spočtenou chybu a použije ji k úpravě svých parametrů algoritmem *backpropagation* a tuto chybu také propaguje i na vstupní neurony.

Chyby vstupních neuronů jsou poté vstupem učícího algoritmu předchozí vrstvy (např. konvoluční vrstvy), která chybu opět využije k úpravě svých parametrů, pokud nějaké má. A chybu opětovně rozdistribuuje na vstupní neurony, kdy v důsledku provádí „opačnou“ operaci, než při dopředném průchodu vstupu. Tato chyba je pak opět vstupem učení předchozí vrstvy a to se opakuje až do dosažení první vrstvy konvoluční neuronové sítě.

3.4 Stavební bloky

Jak již bylo popsáno, konvoluční neuronové sítě se skládají z různého množství za sebou jdoucích vrstev (existují ale také implementace umožňující grafové uspořádání). Prakticky vždy se vyskytuje alespoň jedna *konvoluční vrstva* a poslední vrstva je většinou *plně propojená vrstva*.

Některé vrstvy charakterizujeme pouze pomocí *hyperparametrů* (nastavuje je člověk, případně algoritmus optimalizující architekturu CNN) a některé také pomocí *parametrů* (síť se je sama učí, například váhy filtrů a plně propojené vrstvy).

V dalších kapitolách budou popsány jednotlivé vrstvy. Na začátek bude vždy popsána jejich funkce a (hyper)parametry a poté popsána takzvaná *forward propagation* – tedy dopředný průchod sítí a *backward propagation* – zpětný průchod sítí, propagace chyby a učení parametrů sítě.

Mimo zde zmíněné existují i další používané vrstvy, avšak ty nejsou součástí knihovny a proto jim zde není věnována taková pozornost. Takové vrstvy jsou typicky součástí rozsáhlejších knihoven, než je knihovna vyvíjená v rámci této diplomové práce.

V odstavcích níže bude k buňkám vstupní a výstupní třírozměrné matice pro jednoduchost odkazováno jako ke vstupním a výstupním neuronům. Každý neuron tak má třírozměrnou souřadnici v dané matici a výstupní hodnotu dánu hodnotou matice na tomto místě.

3.4.1 Konvoluční vrstva

Konvoluční vrstvy jsou základem CNN. Hlavní částí jsou třírozměrné filtry, jejichž šířka a výška je typicky malá (mnohem menší než rozměry vstupu), ale hloubka je stejná jako u vstupu.

Předmětem učení jsou váhy filtrů. Hyperparametry nám pak udávají počet filtrů, jejich šířku i výšku (typicky stejné) a krok, se kterým je posouváme po vstupní matici. Často je také možno určit i tzv. *zero padding*, který okraje obrázku doplní nulami a to proto, aby vstup i výstup měli stejnou výšku a šířku.

Vstupních hyperparametrů je tedy pět:

- počet filtrů K – každý filtr je třírozměrný, udává hloubku výstupu,
- rozměr filtrů F – šířka a výška filtru,
- krok S – velikost posunutí filtru po vstupu,
- počet okrajových nul P – počet nul na okraji, typicky 0 (nazývané *valid padding*) nebo taková hodnota, aby šířka a výška vstupu byly zachovány (nazývané *same padding*, $P = \frac{F-1}{2}$),
- zda využít *bias* nebo ne.

V případě vstupní velikosti $W_{in} \times H_{in} \times D_{in}$, kde W značí šířku, H výšku, D hloubku, lze spočítat výstupní velikost[1] jako:

$$W_{out} = \frac{W_{in} - F + 2P}{S + 1} \tag{3.1}$$

$$H_{out} = \frac{H_{in} - F + 2P}{S + 1} \tag{3.2}$$

$$D_{out} = K \tag{3.3}$$

To celé nám v konečném součtu dává $(F \times F \times D_{in} + 1) \times K$ vah, kde jednička značí *bias* (v případě konvoluční vrstvy neupravujeme váhu hrany směřující k biasu, ale přímo jeho hodnotu).

Pokud si představíme, že by se jednalo o plně propojenou neuronovou síť, pak počet parametrů by byl zhruba $W_{in} \times H_{in} \times D_{in} \times W_{out} \times H_{out} \times K$, kde navíc ještě nejsou zahrnuty váhy vedoucí k *bias* neuronům. Redukce počtu parametrů je tedy markantní.

Dopředné šíření Filtr postupně posouváme po vstupním obrázku a obsah filtru násobíme se vstupem na stejných pozicích a sčítáme dohromady. Každé takové posunutí produkuje jednu akumulovanou výstupní hodnotu. V případě, že je využit bias, je pak k této hodnotě přičtena hodnota biasu. Každý filtr tak produkuje dvourozměrnou výstupní matici. Hloubka výstupu je dána počtem filtrů.

Jak již bylo zmíněno, operace konvoluce v pravém významu předpokládá „otočený“ filtr, což není příliš intuitivní. Často se tedy provádí spíše operace nazývaná *cross-correlation*, u které filtr není otočen, jak lze vidět na obrázku 3.2.

0	0	0	0	0	0
0	1	2	3	4	0
0	5	6	7	8	0
0	9	10	11	12	0
0	13	14	15	16	0
0	0	0	0	0	0

$\times$

1	0	2
3	2	1
1	2	1

$\Longrightarrow$

20	30	44	32
49	71	94	75
80	135	146	111
60	113	122	80

Obrázek 3.2: Vizualizace průchodu konvoluční vrstvou pro $F = 3$, $S = 1$, $P = 1$, $K = 1$ a bez použití *bias* neuronu, vstup má hloubku 1 (stejně tedy i filtry)

Zpětné šíření V případě konvoluční vrstvy dochází k učení hodnot filtrů (tyto hodnoty často nazýváme váhami) a hodnoty *bias* neuronů. K tomu se, stejně jako v případě neuronových sítí, využívá zpětného šíření chyby a gradientního sestupu.

Jak bylo řečeno, tak každé vrstvě výstupu (dle hloubky) odpovídá jeden filtr, kdy všechny váhy filtru se podílí na každé chybě v této vrstvě. Pro každou váhu filtru je tedy hodnota, o kterou jej budeme upravovat zvoleným optimalizátorem, dána chybou na výstupním neuronu a také příslušným vstupním neuronem – je tedy nutné zjistit vstupní hodnotu pro tuto konkrétní váhu při posunutí, které produkovalo konkrétní chybu. To lze realizovat mapovací funkcí, která vypočítá souřadnice tohoto vstupního neuronu.

[1] Zde je na místě poznamenat, že šířka i výška samozřejmě musí být celé číslo. Pokud vychází číslo s nenulovou hodnotou za desetinnou čárkou, pak jsou hyperparametry nevhodně zvoleny.

Co se týče hodnoty *bias* neuronu, tak ten se podílel na všech chybách v dané vrstvě výstupní matice a každá chyba je tak akumulována a využita pro jeho pozdější úpravu.

Chybu je dále nutno také propagovat na vstupní neurony. Za tímto účelem je potřeba pro každou chybu zjistit, jaké vstupy se na ní podíleli (viz výše) a s jakou vahou (dáno váhou filtru v dané pozici). Jeden vstupní neuron se může podílet na více chybách vzhledem k tomu, že posouvání filtru se může překrývat a taktéž proto, že jeden vstupní neuron je použit opakovaně ve všech vrstvách výstupní trojrozměrné matice. Tato váhovaná chyba se pro každý vstupní neuron akumuluje a poté se využívá pro učení předcházející vrstvy.

Princip je tedy stejný jako v případě algoritmu 1, avšak komplikovanější vzhledem k tomu, že není tak přímočaré zjistit (v závislosti na implementaci), které váhy a vstupy se podílely na konkrétní chybě.

3.4.2 Poolingová vrstva

Poolingová vrstva má za cíl snižovat velikost vstupu, což ve výsledku znamená, že následné operace mají menší paměťovou a také časovou náročnost. Mimo to tato vrstva také zavádí určitou nezávislost na posuvu a rotaci. Ovlivněna je pouze šířka a výška, na hloubku operace nemá vliv, neboť pracuje nezávisle po jednotlivých vrstvách.

Vstupní hyperparametry jsou tři:

- rozměr okna F – velikost okna, uvnitř kterého se provádí vybraná operace,
- krok S – velikost posunu okna,
- typ operace – typ prováděné operace (nejčastěji *Max* nebo *Average pooling*).

V případě vstupní velikosti $W_{in} \times H_{in} \times D_{in}$, kde W značí šířku, H výšku, D hloubku, lze spočítat výstupní velikost jako:

$$W_{out} = \frac{W_{in} - F}{S + 1} \tag{3.4}$$

$$H_{out} = \frac{H_{in} - F}{S + 1} \tag{3.5}$$

$$D_{out} = D_{in} \tag{3.6}$$

Dopředné šíření Pro všechny vrstvy (dle hloubky) je postupně po řádcích posouváno okno velikosti F s krokem S (jak horizontálně, tak vertikálně) a nad obsahem okna je vždy provedena vybraná operace, jejíž výsledek je pak vložen na odpovídající výstupní místo.

Nejčastější operací je *Max pooling*, kdy je vybrána maximální hodnota. Často používanou je i operace *Average pooling*, kdy výstupní hodnota je dána průměrem hodnot v okně.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$\Longrightarrow$

6	8
14	16

Obrázek 3.3: Vizualizace *Max pooling* operace při $F = 2$ a $S = 2$

Co se týče velikosti okna a kroku, tak se velmi často volí $F = 2, S = 2$ nebo $F = 3, S = 2$ (tzv. *překrývající se pooling*).

Zpětné šíření Všechny parametry této vrstvy jsou dány již návrhem a nedochází k učení. Vrstva ale samozřejmě musí propagovat vzniklou chybu na předchozí vrstvy. Způsob této propagace závisí na zvolené operaci.

V případě *max poolingu* je chyba přiřazena vstupu, který byl v posunutém okně, udávajícím výstupní hodnotu, největší. V případě *average poolingu*, je chyba rovnoměrně rozdistribuována na všechny vstupy v okně. Chyby se na vstupech sčítají (jedna vstupní hodnota totiž může být součástí více oken, pokud se jedná o již zmíněný překrývající se pooling).

Budoucnost Je možné, že v budoucnu se tato vrstva prakticky nebude využívat, protože některé studie navrhují spíše využívání konvolučních vrstev s větším krokem namísto poolingových vrstev [37].

Zatím ale stále převládá názor, že konvoluční vrstvy by měly provádět pouze hloubkovou transformaci vstupu, nikoliv měnit jeho výšku či šířku. I proto se typicky u konvolučních vrstev používá $S = 1$ a doplnění nulami takové, aby rozměry zůstaly totožné.

3.4.3 Aktivační vrstva

Aktivační vrstva má stejný účel jako aktivační funkce v klasických neuronových sítí. Je však časté, že v konvolučních neuronových sítích není přímo součástí vrstev (typicky konvoluční a plně propojené), ale zařazuje se až následně jako samostatná vrstva. Stejně tomu bude v této práci.

V případě konvolučních vrstev je často využívána aktivační funkce *ReLU* či případně *Leaky ReLU*. V případě plně propojených vrstev pak i další funkce (v případě klasifikace je časté, že poslední vrstva má aktivační funkci *Softmax*).

Vrstva nemá žádné parametry a hyperparametrem je pouze typ aktivační funkce, která je aplikována na vstupy.

Dopředné šíření Průchod aktivační vrstvou je velmi prostý, nad všemi vstupními hodnotami je provedena daná operace. Jako příklad zde uvažujme operaci *ReLU* danou vzorcem:

$$f(x) = \begin{cases} x & \text{pro x} \geq 0 \\ 0 & \text{pro x} < 0 \end{cases} \tag{3.7}$$

1	2	-3	4
5	-6	-7	8
9	10	-11	12
13	14	-15	-16

$\Longrightarrow$

1	2	0	4
5	0	0	8
9	10	0	12
13	14	0	0

Obrázek 3.4: Vizualizace provedení *ReLU* operace

Zpětné šíření Tato vrstva se opět žádné parametry neučí, ale musí propagovat chybu na vrstvy předchozí. Je tedy nutno provést derivaci aktivační funkce, jejímž výsledkem je pak vynásobena chyba výstupního neuronu a přiřazena vstupnímu neuronu.

Například v případě již zmíněné funkce *ReLU* pouze dochází k tomu, že vynulované vstupy (tedy ty menší než nula) žádnou chybu nemají přiřazenu, zbylé vstupy chybu dostanou přiřazenu beze změny (násobeno jednou).

3.4.4 Drop-out vrstva

Problémem neuronových sítí s velkým množstvím parametrů je možnost přeučení (*overfitting*) a ztráta schopnosti generalizovat. Jednou z možných forem boje proti tomuto problému je využití takzvaných *drop-out* vrstev. Ty mají svůj účel pouze během trénování a v natrénované síti poté chybí.

Motivací za jejich použitím je úprava vstupu pro následující vrstvu tak, aby byl pokaždé jiný (a nemohlo tedy dojít k jeho „přesnému“ naučení, taková síť je pak také robustnější a lépe generalizuje [23]). Toho je dosaženo náhodným nulováním vstupních hodnot během dopředného průchodu. Při zpětném průchodu pak vynulované vstupní neurony dále nepropagují chybu, ostatní ji propagují nezměněnou.

V této vrstvě uvažujeme následující hyperparametry:

- *drop-out rate* – pravděpodobnost vynulování hodnoty (typicky nižší hodnota z intervalu $\langle 0, 1 \rangle$).

3.4.5 Plně propojená vrstva

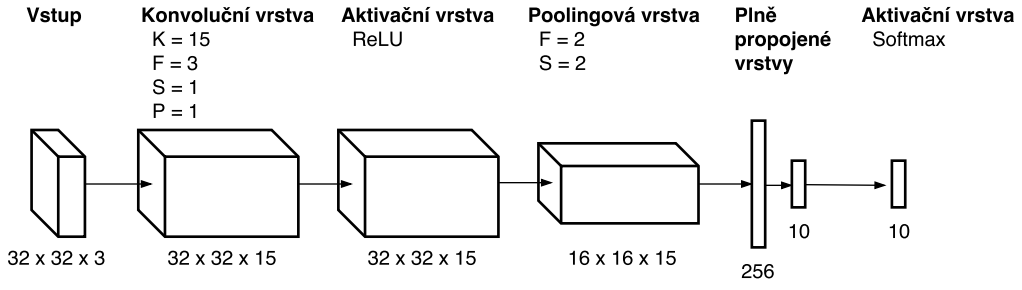
Plně propojené vrstvy (anglicky *fully connected layers*) se typicky umisťují na konec konvoluční neuronové sítě (i několik takových vrstev za sebou), kde je výstup předchozí třídimenzionální vrstvy linearizován do jednodimenzionálního pole a výstup poslední plně propojené vrstvy je pak celkovým výstupem konvoluční neuronové sítě.

V zásadě se jedná o obyčejnou plně propojenou neuronovou síť bez skrytých vrstev. V tomto případě uvažujeme následující hyperparametry:

- velikost výstupní vrstvy – v případě poslední vrstvy dáno řešenou úlohu, ve zbylých případech určeno programátorem,
- zda využít *bias* neuronů nebo ne.

3.5 Architektura sítě

Architektura konvoluční neuronové sítě velmi závisí na zvolené úloze. Na internetu lze dohledat *state-of-the-art* architektury pro různé úlohy, o žádné takové architektuře ale nelze prohlásit, že by byla univerzální. I přesto je ale doporučeným postupem nalézt architekturu, která již funguje (ideálně i s předtrénovanými váhami) a doladit ji na vlastních datech.



Obrázek 3.5: Ukázková architektura pro řešení problému se vstupem 32×32 pixelů o třech kanálech a s deseti výstupními třídami (odpovídá CIFAR-10) [22]

Existují sice snahy o vytvoření univerzálních architektur, ale ty jsou pak v důsledku velmi rozsáhlé a jejich učení trvá příliš dlouhou dobu. Jejich použití na jednodušší úlohy je zbytečné.

Typicky však platí následující pravidla:

- první vrstvou je vstupní vrstva, která vstup nijak netransformuje,
- poslední vrstvou je plně propojená vrstva, kde počet neuronů výstupní vrstvy neuronové sítě odpovídá počtu tříd (nebo počtu požadovaných výstupních hodnot),
- mezi první a poslední vrstvou se vyskytují konvoluční vrstvy, často následované aktivačními vrstvami,
- po jedné nebo více konvolučních vrstvách (a s tím spojených aktivačních vrstvách) se vyskytují poolingové vrstvy.

Takovou architekturu konvoluční neuronové sítě pak můžeme popsat následujícím regulárním výrazem 3.8 [21], kde I značí vstupní vrstvu, C konvoluční vrstvu, A aktivační vrstvu, P poolingovou vrstvu, D dropout vrstvu a F plně propojenou vrstvu.

$$I\,(\,(D?\,C\,A?\,)+\,P\,)+\,(\,D?\,F\,A?)+ \tag{3.8}$$

Co se týče nastavení hyperparametrů jednotlivých vrstev, tak obecná pravidla jsou následující [21]:

- rozměry vstupu by měly být dělitelné dvěma,
- filtry konvolučních vrstev by měly mít malou velikost (například 3×3 nebo 5×5) a krok 1,
  - výjimkou může být první konvoluční vrstva, která za účelem snížení paměťových, ale hlavně časových, nároků může obsahovat filtr většího rozsahu, který je posouván s větším krokem, například 11×11 s krokem 4,
- doplnění nulami se často u konvolučních vrstev nastavuje tak, aby byla zachována šířka a výška vstupu,
- počet filtrů je obtížné určit, často se ale jedná o mocniny dvou – $16, 32, 64, 128, \ldots$,
- poolingová vrstva má typicky rozsah 2 a krok 2, což zmenší vstup o 75%, jiná nastavení nejsou příliš obvyklá (větší rozsah by znamenal již příliš velkou ztrátu dat).

3.5.1 Příklady architektur

Existuje několik architektur, o kterých lze často najít zmínku v literatuře [21]. Některé z těchto architektur budou dále popsány.

LeNet Jedna z prvních úspěšných aplikací konvolučních neuronových sítí v praxi. Jedná se o několik architektur (například *LeNet-5* využívaná pro rozpoznání číslic v obraze [24], která obsahuje tři konvoluční vrstvy) vyvinutých *Yann LeCunem* v 90. letech 20. století.

AlexNet Další z velmi významných neuronových sítí, která zažehla zájem o využití konvolučních neuronových sítí pro zpracování obrazu. Architektura je mnohem složitější než v případě *LeNet*. Vyhrála soutěž ILSVRC (*ImageNet Large Scale Visual Recognition Competition* [35]) v roce 2012 a to s velkým náskokem 10% na druhé místo.

ZFNet *ZFNet* byla vyvinuta *Matthew Zeilerem* a *Robem Fergusem*. Prakticky se jedná o *AlexNet* avšak s pozměněnými hyperparametry konvolučních vrstev (zmenšení filtrů a zvětšení jejich počtu, tedy více parametrů a s tím související vyšší přesnost ale i delší doba trénování).

GoogLeNet Vítěz ILSVRC z roku 2014. Tato architektura přišla s novými nápady – zejména s tzv. *inception* moduly, které významně redukují počet parametrů sítě, a také používá *average pooling* místo některých plně propojených vrstev.

VGGNet VGGNet poukázala na fakt, že počet vrstev konvoluční neuronové sítě má velký dopad na její úspěšnost. Celkem obsahovala 16 konvolučních a plně propojených vrstev (a další poolingové), které prováděly jen jednoduché operace. I přesto se ale dokázala umístit hned za *GoogLeNet*.

Nevýhodou této architektury je fakt, že obsahuje velké množství parametrů. Později navíc bylo zjištěno, že některé plně propojené vrstvy je možno vyloučit bez dopadu na úspěšnost (což přispívá k významné redukci počtu parametrů) [21].

ResNet Architektura *ResNet* [17] vyhrála soutěž ILSVRC v roce 2015. Oproti jiným používá speciální *skip connections* (zlepšují tok gradientů skrze vrstvy sítě během trénování), *batch normalization* (obdoba předzpracování vstupních dat, avšak využíváno mezi vrstvami sítě) a neobsahuje plně propojenou vrstvu na konci. *ResNet* aktuálně patří mezi jedny z nejlepších architektur pro zpracování obrazu a její varianty jsou často používány jako defaultní při aplikaci konvolučních neuronových sítí v praxi.

Kapitola 4

Návrh knihovny

Na základě získaných znalostí bude v této kapitole navržena knihovna pro práci s konvolučními neuronovými sítěmi. Ta musí obsahovat všechny často používané vrstvy, jejich propojení a trénování. Aby bylo použití knihovny efektivní a práce s ní jednoduchá, je nutné, aby knihovna obsahovala i další moduly, které práci usnadní (například ukládání natrénované sítě, načítání vstupních dat a další).

Návrh pracuje s předpokladem, že následná implementace bude provedena v objektově orientovaném jazyce (přesněji v jazyce `C++`, jehož volba bude zdůvodněna v kapitole 5).

Knihovna byla pojmenována *TypeCNN* s odkazem na její přístup k datovým typům, což bude rozvedeno dále v kapitole 4.3.

4.1 Konvoluční neuronová síť

Jádrem knihovny je samozřejmě konvoluční neuronová síť. Ta v sobě ale prakticky obsahuje i implementaci plně propojené dopředné neuronové sítě ve formě několika plně propojených vrstev následovaných aktivačními vrstvami. Není tedy problém, aby tato knihovna zároveň vystupovala i jako knihovna pro tento typ umělých neuronových sítí. Uživateli jsou tedy nabízena dvě hlavní rozhraní – `ConvolutionalNeuralNetwork` a `NeuralNetwork`.

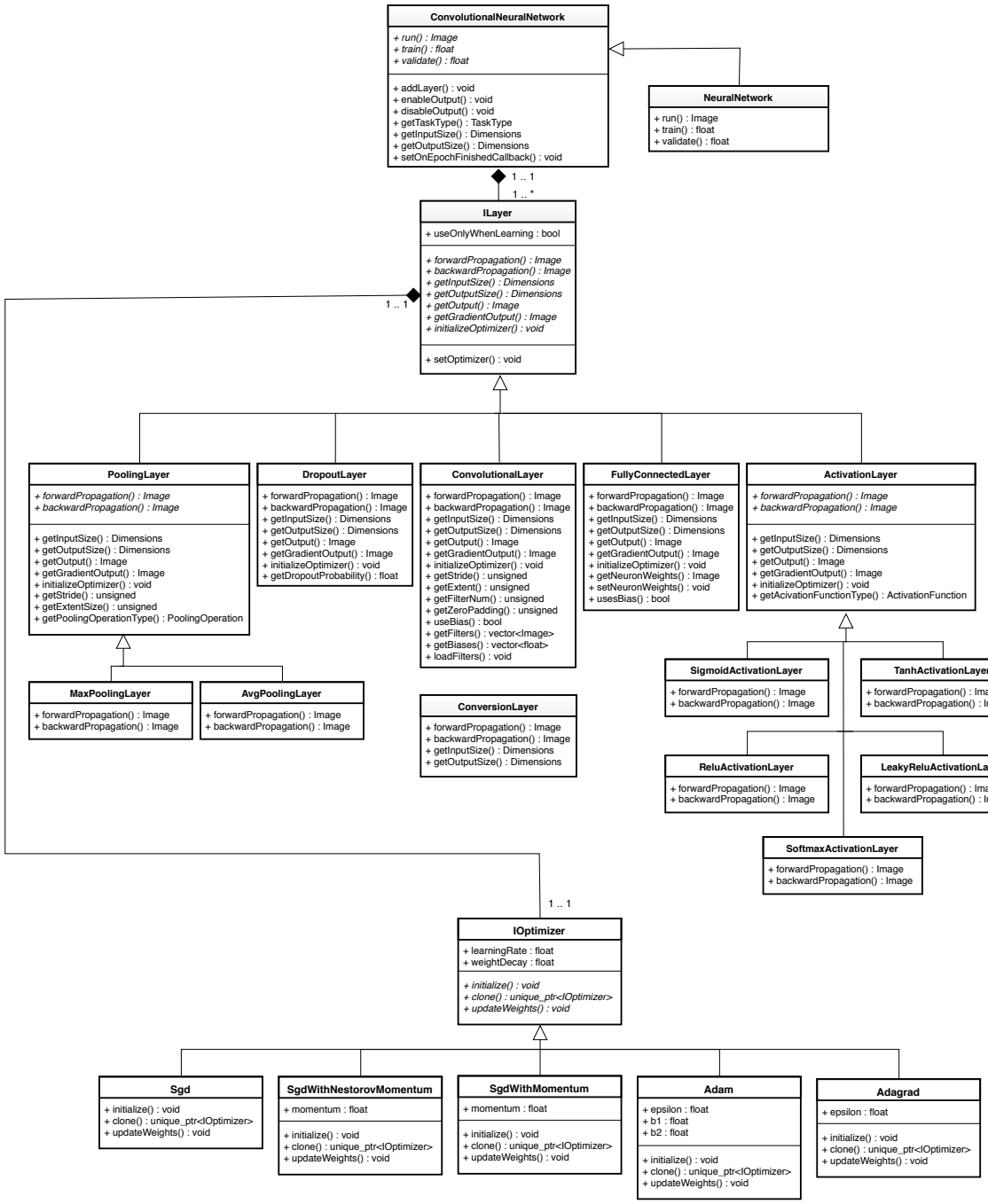
Na obrázku 4.1 lze vidět diagram tříd jádra knihovny, diagram zbylých tříd lze nalézt na obrázku 4.2. Vyobrazeny jsou zároveň také metody vyvedené na jejich rozhraní (kromě *konstruktorů*) u kterých je uveden návratový typ. Chybí typy parametrů a privátní metody za účelem zachování přehlednosti diagramů a také vzhledem k tomu, že se během implementace často mění a je obtížné postihnout všechny závislosti během návrhu.

I přesto, že v diagramu lze vidět datový typ `float` pro váhy a vstupní/výstupní hodnoty, tak datovým typem ve skutečnosti bude *typový alias*, který bude možno jednoduše měnit. Síť bude obsahovat celkem tři aliasy – typ vah, typ inference a typ trénování (viz kapitola 4.3).

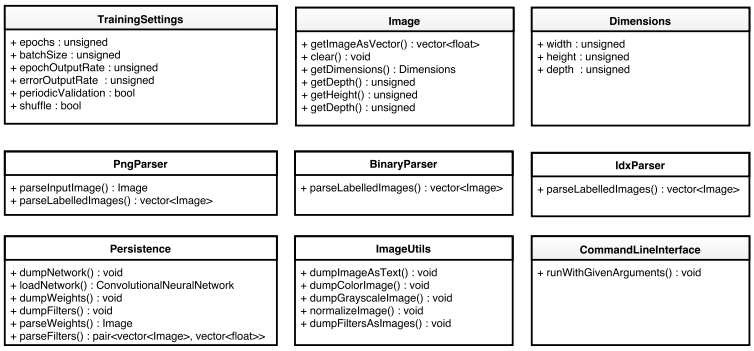
4.1.1 Rozhraní

Jádrem konvoluční neuronové sítě je třída `ConvolutionalNeuralNetwork`, kterou by jako jedinou mělo být třeba instanciovat při využívání knihovny (s výjimkou pomocných tříd, viz kapitola 4.2). Obsahuje jak metody pro inferenci – `run()`, tak pro trénování – `train()` a validaci – `validate()`. Samozřejmostí je také přidávání nových vrstev – `addLayer()`.

Pro účely úpravy parametrů a sledování průběhu trénování je také vhodné, aby konvoluční neuronová síť mohla periodicky podávat uživateli informace o stavu trénování. Proto



Obrázek 4.1: Diagram tříd pro jádro knihovny



Obrázek 4.2: Diagram pomocných tříd a modulů

je také možno specifikovat *callback* funkci metodou `setOnEpochFinishedCallback()`, která bude provedena po každé epoše. V rámci funkce by také mělo jít měnit parametry trénování (třída `TrainingSettings`, častá je změna učícího koeficientu η).

Další metody jsou pak spíše jen okrajovou záležitostí, jedná se o metody, které používá zbytek modulů knihovny.

Jak již bylo zmíněno, tak k dispozici je také třída `NeuralNetwork`, která místo třírozměrných matic přijímá jako vstup jednorozměrné vektory. Jelikož samotné neuronové sítě nejsou středem zájmu v této práci, jedná se o adaptér ke třídě `ConvolutionalNeuralNetwork`.

4.1.2 Vrstvy

Podporovány jsou všechny dříve zmíněné vrstvy (viz kapitola 3.4), každá vrstva je reprezentována vlastní třídou. Celkově se jedná o třídy: `ConvolutionalLayer`, `PoolingLayer`, `ActivationLayer`, `DropoutLayer` a `FullyConnectedLayer`. Každá vrstva má své rozhraní pevně dáno abstraktní třídou `ILayer`, která vynucuje implementaci následujících metod:

- `forwardPropagation()` – provádí dopředný průchod vrstvou, parametry je vstupní matice a reference na výstupní matici, tento průchod je využíván jak během inference, tak během trénování,
- `backwardPropagation()` – provádí zpětný průchod vrstvou, parametry jsou vstup a výstup dopředného průchodu, vstupní gradienty, reference na matici výstupních gradientů a také parametry učení, tento průchod je využíván pouze během učení a upravuje parametry vrstvy,
- `getInputSize()` – vrací očekávanou velikost vstupní trojrozměrné matice,
- `getOutputSize()` – vrací velikost výstupní trojrozměrné matice,
- `getOutput()` – vrací referenci na privátní matici pro uložení výstupu (aby nebylo nutné ji uchovávat jako proměnnou),
- `getGradientOutput()` – vrací referenci na privátní matici pro uložení gradientů, které slouží jako vstup učení předcházející vrstvy,
- `initializeOptimizer()` – nutno implementovat pro vrstvy provádějící učení parametrů, slouží k inicializaci vnitřních struktur optimalizátoru (viz dále).

Prvotní návrh uvažoval, že dopředný a zpětný průchod bude mít jako své návratové hodnoty matice výstupních hodnot (výstup prováděné operace pro dopředný průchod, gradienty pro předchozí vrstvu v případě zpětného průchodu). Od tohoto přístupu ale bylo upuštěno, neboť jednak zpomaluje běh programu (nutná opakovaná instanciace a inicializace třídy `Image`) a také je málo flexibilní. Matice jsou tedy předávány jako parametr metody. Pro normálního uživatele jsou tyto metody skryty, takže přidaná komplexita není problémem.

Třídy `ConvolutionalLayer`, `DropoutLayer` a `FullyConnectedLayer` implementují veškeré metody rozhraní. Další třídy poskytují pouze částečnou implementaci metod a dále se dělí dle typu prováděné operace. K tomuto kroku bylo přistoupeno s ohledem na časovou efektivitu – lze se tak vyhnout provádění zbytečných kroků (například konstrukce `if` nebo `switch`) souvisejících s tím, že není dopředu jasné, jaká konkrétní operace bude prováděna.

PoolingLayer Třída `PoolingLayer` ponechává metody dopředného a zpětného průchodu abstraktní. Jejími podtřídami jsou `AvgPoolingLayer` a `MaxPoolingLayer`, jedná se o dvě nejčastější operace v rámci *poolingové* vrstvy.

ActivationLayer Jak již bylo zmíněno, tak žádná vrstva přímo neobsahuje aktivační funkce, ale aktivační funkce jsou reprezentovány vlastní vrstvou. Takové řešení je sice o něco málo pomalejší z hlediska výkonnosti, ale za to flexibilnější.

Tato vrstva se nazývá `ActivationLayer` a taktéž poskytuje částečnou implementaci metod rozhraní – s výjimkou dopředného a zpětného průchodu, který je specifický pro každou aktivační funkci. Jejími podtřídami jsou různé aktivační funkce, kterých je podporováno celkem pět – `LeakyReluActivationLayer`, `ReluActivationLayer`, `SigmoidActivationLayer`, `SoftmaxActivationLayer` a `TanhActivationLayer` (viz kapitola 2.4.4).

ConversionLayer Speciální vrstvou je vrstva `ConversionLayer`, která provádí převod mezi dvěma následujícími vrstvami s různými datovými typy pro inferenci, mezi kterými neexistuje implicitní převodní funkce. Důvod jejího použití bude popsán dále v kapitole 5.4. I když vrstva poskytuje obdobné metody jako ostatní, tak vzhledem k existenci více datových typů nemůže dědit rozhraní `ILayer`.

4.1.3 Uložení dat

Vstupy a výstupy vrstev, váhy a další parametry, které je možno vyjádřit jako matice, jsou reprezentovány třídou `Image`. Ta je primárně určena pro uložení třídimenzionálního pole hodnot, avšak data jsou uložena po řádcích v jednorozměrném poli (což je jednodušší, paměťově méně náročné a tento linearizovaný formát lze využít pro zrychlení operací, pro které není podstatná struktura).

Vzhledem k výše zmíněnému lze tedy třídu využít i pro uložení jedno a dvou dimenzionálních dat. Přístup k datům je dán mapovacími funkcemi, ty jsou celkem tři – pro 1D, 2D a 3D. Stejně tak je možno instance třídy `Image` vytvářet více konstruktory z různých formátů (pro jednoduchost použití potencionálními uživateli, ale například i v testech).

Rozměry uložené matice jsou dány strukturou `Dimensions`, která definuje její výšku, šířku a hloubku. Pro 2D matice je hloubka 1, pro 1D matice je šířka i hloubka 1.

4.1.4 Trénování

Složitější částí bude implementace učícího mechanismu konvoluční neuronové sítě a zejména jeho náležité otestování a validace. Trénování bude založeno na algoritmu *zpětného šíření chyby* (anglicky *backpropagation*), který u učení neuronových a konvolučních neuronových sítí převládá.

Trénování sítí je časově velmi náročnou činností (zejména na *CPU*). Je samozřejmě důležité, aby učení bylo implementováno bez chyb, ale také velmi důležité, aby bylo dostatečně efektivní. Například operace konvoluce nebo úpravy vah se provádí velmi často a každé, i minimální, vylepšení algoritmu má ve výsledku velký vliv na celkovou časovou náročnost.

Optimalizací existuje velké množství. V případě této knihovny bude kladen důraz především na efektivitu algoritmů. Vůbec největším urychlením je implementace pro grafické procesory, té však v této práci nebude využito vzhledem k časovým možnostem a také vzhledem k rozšíření založeném na nezávislosti na datových typech.

Optimalizátory

Ve velké části menších knihoven nalezneme pouze jeden typ přístupu k úpravě vah pomocí gradientního sestupu. A to buď ten základní, který využívá pouze učící koeficient anebo o něco pokročilejší, který využívá také momentum a případně i regularizaci ve formě *weight decay*.

Vytvořená knihovna by ale měla nabízet více těchto optimalizátorů a také uživateli umožnit jednoduše přidat vlastní. Každý optimalizátor musí splňovat rozhraní `IOptimizer`, vyžadující implementaci pěti metod:

- `initialize()` – inicializuje velikost a obsah datových struktur optimalizátoru (nutné pro optimalizátory, které si uchovávají stav),
- `clone()` – vytvoří kopii optimalizátoru s neinicializovanými strukturami, ale stejnými nastaveními,
- `updateWeights()` – tři metody se stejným názvem, které upravují váhy na základě aktuálních hodnot, *delta* hodnot a vnitřních struktur optimalizátoru.

Metoda `updateWeights()` je přetížena a prováděné operace se liší podle typů parametrů. Váhy pro plně propojenou vrstvu jsou reprezentovány dvourozměrnými maticemi (včetně vah vedoucích k biasům), filtry konvoluční vrstvy jsou reprezentovány polem třírozměrných matic a biasy konvoluční vrstvy jsou reprezentovány polem hodnot.

Knihovna bude v základu podporovat celkem pět optimalizátorů (viz kapitola 2.4.7):

- SGD,
- Momentum,
- Nestorov momentum,
- Adagrad,
- Adam.

První dva jsou si velmi podobné, kdy *SGD* je stejné jako *Momentum* s hodnotou momenta γ rovnou nule, avšak opět došlo k rozdělení s ohledem na rychlost. *SGD* také nepotřebuje ukládat svůj stav. Úprava vah je jednou z nejčastěji prováděných operací, zejména v případě plně propojených vrstev, a i takto prosté zjednodušení může mít velký dopad na rychlost.

4.2 Podpůrné prostředky

Aby bylo možno knihovnu efektivně používat, je také vhodné poskytnout uživateli další podpůrné prostředky. Často se lze setkat s tím, že menší volně dostupné knihovny obsahují pouze samotnou konvoluční neuronovou síť (lze nalézt i takové, které k nim neposkytují žádné rozhraní a implementace mechanismu trénování je tak ponechána na uživateli) a to je vše. Uživatel pak musí sám zajistit načítání dat nebo perzistentní ukládání naučené sítě.

Proto by tato knihovna měla poskytovat tyto dva hlavní podpůrné prostředky – perzistenci a načítání dat. Jejich diagram tříd, společně s dalšími okrajovými moduly a třídami, lze vidět na obrázku 4.2.

Mimo to je také vhodné ke knihovně připravit i konzolové rozhraní, které jednak usnadní její testování, ale také experimentování s různými nastaveními a architekturami (bez nutnosti neustále překládat knihovnu jako v případě využití programového rozhraní).

4.2.1 Perzistence

Perzistence musí umožnit uživateli uložit naučenou síť a poté ji opětovně načíst – a využívat nebo případně i dotrénovat. Architektura sítě bude uložena v jednoduchém XML souboru, jehož struktura byla také předmětem návrhu.

```
<convolutional_neural_network>
    <settings>
        <task type="classification"/>
        <input width="28" height="28" depth="1"/>
        <output width="10" height="1" depth="1"/>
    </settings>
    <architecture>
        <layer type="convolutional">
            <stride value="1"/>
            <zero_padding value="0"/>
            <filters extent="5" number="8" path="1_conv_layer.txt"/>
            <bias use="true"/>
        </layer>
        <layer type="activation">
            <activation type="relu"/>
        </layer>
        <layer type="pooling">
            <operation type="max"/>
            <stride value="2"/>
            <extent value="2"/>
        </layer>
        <layer type="dropout">
            <probability value="0.1"/>
        </layer>
        <layer type="fully_connected">
            <output_layer size="10"/>
            <weights path="5_fc_layer.txt"/>
            <bias use="true"/>
        </layer>
        <layer type="activation">
            <activation type="sigmoid"/>
        </layer>
    </architecture>
</convolutional_neural_network>
```

Ukázka 4.3: Ukázka XML souboru s architekturou konvoluční neuronové sítě se šesti vrstvami

Jak vypadá uložená síť lze vidět v ukázce 4.3. Schéma se skládá ze dvou hlavních částí – `settings` a `architecture`. V `settings` je potřeba zvolit typ úlohy (pro nastavení formátu výstupů a typu výpočtu přesnosti sítě během validace), velikost vstupu a velikost výstupu. V části `architecture` jsou již za sebou řazeny jednotlivé vrstvy konvoluční neuronové sítě reprezentované uzlem `layer`. V závislosti na typu uzlu je pak možno upravit parametry vrstvy.

Klasickou neuronovou síť je možno zadat stejným způsobem – architektura však bude obsahovat pouze uzly typu `fully_connected` a `activation`. Výška a hloubka vstupu, respektive výstupu, se pak nastaví na 1.

Naučené váhy jsou uloženy v separátních textových souborech, na které je odkazováno z XML souboru. Textový formát byl zvolen vzhledem k jeho jednoduchosti, což také umožňuje snadnou analýzu.

4.2.2 Načítání dat

Načítání dat je problematické, neboť existuje spousta různých formátů. V základu bude knihovna obsahovat tři takzvané *parsery* pro tři formáty, těmi jsou:

- `idx` – formát pro uložení multidimenzionálních dat v binární podobě (společně s metadaty popisujícími data) – využito pro MNIST [25],
- `binary` – data uložena binárně, bez metadat (ty musí uživatel znát a předem zadat) – využito pro CIFAR-10 [22],
- `png` – formát `*.png` pro uložení obrázku o čtyřech kanálech (RGBA), často používaný vzhledem k dobrému poměru kvality a velikosti na disku.

V prvních dvou případech bude podporováno načítání pouze celých datových sad (vzhledem k předpokládanému využití jen během trénování), v posledním případě pak bude možno načíst i jednotlivé soubory (vzhledem k možnému využití jak během inference, tak během trénování).

Další parsery si pak v případě nutnosti musí uživatel dodefinovat sám – stačí vytvořit třídu, která načítá data do dvojice instancí třídy `Image`, kde první z dvojice odpovídá vstupu a druhý z dvojice očekávanému výstupu. To platí pro trénování, pro inferenci stačí samozřejmě načíst pouze jednu instanci.

4.2.3 Konzolové rozhraní

Konzolové rozhraní nejen usnadní práci s knihovnou, ale také poskytne potencionálnímu uživateli náhled na používání jádra knihovny (neboť konzolové rozhraní do jisté míry představuje projekt napsaný nad knihovnou).

Konzolové rozhraní v případě této knihovny bude muset obsahovat velké množství parametrů, aby byly pokryty všechny aspekty práce s konvolučními neuronovými sítěmi. Musí být možno načíst natrénovanou i nenatrénovanou síť, nad kterou bude možné provádět inferenci, trénování i validaci. Zejména v případě trénování pak existuje větší množství parametrů, které může uživatel ovlivnit. Většina z nich by tedy měla být dostupná.

V případě programového rozhraní je nežádoucí, aby knihovna produkovala výstupy do konzole. V případě konzolového rozhraní to však není problémem, spíše naopak. Uživatel by měl být během trénování informován o aktuálním stavu nebo o výsledcích validace. Na to je třeba brát ohled během implementace dalších částí knihovny – aby takové výstupy mohly produkovat v případě, že tak budou nastaveny.

4.3 Nezávislost na datovém typu

Jako rozšíření nad rámec zadání diplomové práce a rozšíření knihovny, aby se tak odlišila od jiných volně dostupných, byla zvolena nezávislost na datovém typu.

Implementace by měla být provedena tak, aby jako typ, ve kterém je prováděna inference i trénování konvoluční neuronové sítě, bylo možno zvolit libovolný datový typ včetně těch uživatelem definovaných. Stejně tak je tedy možno i definovat vlastní operace – a zkoušet tak například vliv aproximovaných funkcí (např. násobení) na přesnost natrénované sítě.

V důsledku tak budou v knihovně existovat tři na sobě nezávislé datové typy – pro váhy, pro inferenci a pro trénování. Tato skutečnost se příliš neprojeví z hlediska návrhu, avšak zásadně ovlivní implementaci.

Požadavky jsou následující:

- uživatelské rozhraní bude obsahovat tři typové aliasy (váhy, inference a trénování), které bude nutno nastavit během kompilace (defaultním typem bude datový typ s plovoucí řádovou čárkou na 32 bitech),
- typ pro trénování bude totožný pro všechny vrstvy,
- typy pro inferenci a váhy bude možno specifikovat pro každou vrstvu zvlášť (to však může vyžadovat speciální implementaci rozhraní uživatelem, viz kapitola 6),
- inference bude veškeré výpočty provádět ve svém specifikovaném datovém typu (tedy všechny mezivýsledky budou uloženy v tomto typu, nikoliv jen výstupy vrstvy),
- zpětný průchod bude všechny výpočty provádět ve svém specifikovaném typu (tedy práce s gradienty, úprava vah atd.),
- váhy budou uloženy v datovém typu pro trénování a do svého specifikovaného datového typu budou převedeny v případě použití během inference anebo při ukládání natrénované sítě na disk,
- implementace by měla být provedena takovým způsobem, aby bylo možno za typový alias dosadit libovolný datový typ (včetně uživatelských) bez nutnosti větších zásahů do kódu (defaultně podporovány však budou pouze datové typy s plovoucí řádovou čárkou a datový typ zmíněný v kapitole 4.3.1).

4.3.1 Případová studie

Pokud bychom používali pouze datové typy s plovoucí řádovou čárkou, avšak jinou šířkou, tak by rozšíření nebylo příliš zajímavé (například 64 bitový typ s plovoucí řádovou čárkou přináší pouze drobné zlepšení oproti typu s 32 bity). Proto v rámci vyhodnocení úspěšnosti implementace tohoto rozšíření bude součástí i případová studie, jejímž cílem bude využití reprezentace s pevnou řádovou čárkou. A to s různými nastaveními počtu bitů před a za řádovou čárkou (a také různých bitových šířek pro inferenci a váhy). Operace nad těmito typy budou prováděny standardním způsobem (tedy bez aproximovaných násobiček atd.).

Cílem bude především demonstrovat funkčnost typové nezávislosti, ale také ukázat, že konvoluční neuronové sítě lze v této reprezentaci provádět i s malým (nebo dokonce žádným) vlivem na přesnost.

Motivací za tímto rozšířením je provádění konvolučních neuronových sítí ve vestavěných zařízeních. Jednoduchá vestavěná zařízení totiž často nedisponují jednotkami pro práci

s reprezentací s plovoucí řádovou čárkou (anglicky *floating point units – FPU*). To platí zejména pro zařízení, pro které je stěžejní co nejnižší spotřeba.

Provádění konvolučních neuronových sítí v reprezentaci s pevnou řádovou čárkou potencionálně (v závislosti na zvoleném zařízení) skýtá následující výhody [12]:

- výpočet v reprezentaci s pevnou řádovou čárkou místo plovoucí řádové čárky může být daleko rychlejší,
- odstranění závislosti na výpočetních jednotkách s plovoucí řádovou čárkou může vést k velkému snížení spotřeby elektrické energie,
- reprezentace vah na bitově užším datovém typu také může velmi zásadně snížit místo na disku nutné pro uložení natrénovaných parametrů (při uložení vah na 8 bitech oproti obvyklým 32 bitům je tato úspora čtyřnásobná, což u velkých sítí znamená i desítky `MB`).

Tyto výhody jsou pak ještě více patrné s klesající bitovou šířkou [29].

Na tomto místě je také vhodné podotknout, že se bude jednat o simulaci výpočtu v pevné řádové čárce – nelze tedy očekávat, že chování bude stejné, jako by bylo v případě použití ve vestavěném zařízení. Například nemusí dojít k úspoře času, neboť zvolená reprezentace a operace nad ní na klasickém počítači nejsou tak efektivní, jako by mohly být v hardwaru. I proto bude cílem experimentů pouze zjistit vliv na přesnost, nikoliv na čas.

Ukázková implementace datového typu `FixedPoint`

Pro tyto účely bude součástí knihovny datový typ nazvaný `FixedPoint`, který bude na celočíselném datovém typu implementovat čísla s pevnou řádovou čárkou (pomocí bitových posuvů a masek). Uživatel si bude moci zvolit nejen celkovou šířku, ale i počet bitů před řádovou čárkou a počet bitů za řádovou čárkou.

Datový typ bude implementovat všechny potřebné operace obvyklým způsobem. Dále je nutné podotknout, že datový typ bude využívat *saturaci* – pokud by v případě výpočtů došlo k přetečení nebo podtečení (*overflow* či *underflow*), tak bude výsledná hodnota rovna maximální, respektive minimální, hodnotě v dané reprezentaci.

Vliv aproximace na konvoluční neuronové sítě

Výhodou konvolučních neuronových sítí (a obecně neuronových sítí) je, že se dokáží velmi dobře vyrovnat se šumem ve vstupních datech. Tímto šumem myslíme například jiná natočení nebo jiná pozadí klasifikovaných entit.

Uvažujme ale nyní, že šum nebude pocházet ze vstupů. Převedení sítě do reprezentace s pevnou řádovou čárkou se projeví taktéž přidáním šumu, se kterým by se konvoluční neuronová síť měla být schopna vyrovnat. Čím menší počet bitů, tím razantnější aproximace, více šumu a s tím související nižší přesnost sítě.

V závislosti na kvalitě aproximace lze samozřejmě očekávat snížení přesnosti, avšak pokud je aproximace vhodná, tak takové snížení nemusí mít zásadní vliv a síť může být možno v této aproximované reprezentaci stále používat v praxi.

Předpokládané využití je takové, že síť bude trénována na datovém typu `float`, poté bude typ dopředného průběhu a vah změněn na `FixedPoint` a proběhne dotrénování. Dotrénování je nutné, neboť samotné převedení vah typicky způsobí pokles v úspěšnosti. Velikost poklesu závisí zejména na počtu bitů za řádovou, ale i před řádovou, čárkou.

Trénování Pro práci s gradienty a vahami je nutná vysoká přesnost, v tomto případě je tedy vhodné používat pouze datový typ s plovoucí řádovou čárkou. To mimo jiné znamená, že je třeba mít váhy uloženy ve dvou reprezentacích – přesnější pro trénování a aproximované pro inferenci, jak bylo zmíněno v požadavcích.

I přesto knihovna změnu datového typu pro trénování podporuje a pro dostatečný počet bitů před a za řádovou čárkou je možné provádět i trénování v této reprezentaci. Vliv na kvalitu trénování je ale značný.

4.3.2 Pevná řádová čárka

U reprezentace s pevnou řádovou čárkou, využívané v této práci, je minimální a maximální hodnota definována počtem bitů před a za řádovou čárkou. Reprezentaci si lze nejlépe představit na příkladu, kde celá část má šířku 4 bity, stejně jako desetinná část. Pro jednoduchost nejprve uvažujme pouze kladná čísla.

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	0	1	0	1	0

Tabulka 4.1: Reprezentace s pevnou řádovou čárkou [36], tmavá část označuje bity před řádovou čárkou a světlá bity za řádovou čárkou

Na 8 bitech můžeme vyjádřit $2^8 = 256$ hodnot, binární zápis výše odpovídá hodnotě 170. Desetinné číslo, které hodnota v této reprezentaci představuje, lze spočítat jako:

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} = 10.625$$

Jednodušeji si pak lze převod představit jako násobení koeficientem $\frac{1}{2^x}$, kde x odpovídá počtu bitů desetinné části. Což je možno si ověřit:

$$170 \cdot \frac{1}{2^4} = 170 \cdot \frac{1}{16} = 10.625$$

V případě konvolučních neuronových sítí je však nutno uvažovat i záporná desetinná čísla. Stejně jako v případě celých čísel se využívá dvojkového doplňku. Binární zápis výše v tomto případě odpovídá hodnotě -86. Po vynásobení koeficientem $\frac{1}{16}$ pak je odpovídající desetinná hodnota -5.375. Nejmenší číslo, které lze vyjádřit, je -8 (binárně `10000000`) a nejvyšší 7.9375 (binárně `01111111`).

Specifika práce s definovaným typem Práce s tímto datovým typem samozřejmě přináší i problémy, které jsou způsobeny právě ztrátou přesnosti. Proto se tento typ využije spíše jen při inferenční části trénování, ne při samotné úpravě vah a výpočtu gradientů.

Problémy spojené s tímto typem lze rozdělit do dvou druhů – problémy vzniklé v důsledku omezení celé části a problémy vzniklé v důsledku omezení desetinné části.

V případě omezení celé části pod přípustnou mez nastává problém, kdy během výpočtu dochází k častému přetékání maximální, respektive podtékání minimální, hodnoty. To má vysoké dopady na schopnost sítě učit se.

V případě omezení desetinné části dochází ke ztrátě přesnosti (to se projeví zejména při převedení již natrénované sítě). Velmi specifickým problémem je počáteční nastavení vah. Váhy se nastavují na velmi malé hodnoty, aby ihned po spuštění nedošlo k nasycení sítě

(což by bylo problémem zejména u některých aktivačních funkcí – např. *sigmoida*). Pokud je ale tato hodnota velmi nízká, tak při jejím uložení do datového typu s nízkým počtem bitů za řádovou čárkou dochází k převedení této hodnoty na nulu. Síť, která obsahuje pouze nulové váhy, se poté vůbec neučí.

Za účelem vyřešení tohoto problému byl zaveden mechanismus, který počítá vhodný koeficient (≥ 1), kterým jsou všechny náhodně vygenerované počáteční váhy násobeny před převedením do daného typu. Koeficient se volí tak, aby zhruba čtvrtina všech vah byla nastavena na nejnižší možnou hodnotu kromě nuly. I tento nízký počet vah poté stačí pro „nastartování“ trénování, což již zajistí, že jsou váhy vhodně nastaveny. Konvergence této metody je pomalejší, ale umožňuje sítím učit se například i na dvou bitech za řádovou čárkou, což by bez tohoto mechanismu možné nebylo. Mechanismus nemá žádný vliv na větší bitové šířky nebo typy s plovoucí řádovou čárkou.

4.3.3 Kvantizace

Další možnou reprezentací, která však není použita v této knihovně, je kvantizace. Tu lze nalézt v některých knihovnách pro konvoluční neuronové sítě, jak bude zmíněno dále, proto je vhodné ji alespoň představit.

Princip je podobný – celočíselná hodnota je opět převáděna na desetinné číslo. K tomu se opět využívá násobení koeficientem. V tomto případě je však koeficient součástí definice, která dále zahrnuje i posunutí (respektive obě hodnoty lze vyjádřit na základě definice minimální a maximální hodnoty intervalu, který má být vyjádřen).

Mějme tedy minimální hodnotu A a maximální hodnotu B, uvažujme uložení na 8 bitech, kde minimální hodnota je -128 a maximální 127. Pro výpočet desetinného čísla x z celého čísla y lze použít následující rovnice:

$$
\begin{aligned}
x &= C\,(y - D) \\
D &= -128 - \frac{A}{C} \\
C &= \frac{B - A}{127 - (-128)}
\end{aligned}
$$

C vyjadřuje škálovací hodnotu a D vyjadřuje hodnotu nulového bodu. Pokud tedy budeme uvažovat předcházející příklad opět na 8 bitech, kdy minimum je -8, maximum 7.9375 a hodnota celého čísla -86, tak lze desetinnou hodnotu spočítat následovně:

$$
\begin{aligned}
C &= \frac{7.9375 - (-8)}{127 - (-128)} = 0.0625 \\
D &= -128 - \frac{-8}{C} = -128 + \frac{8}{0.0625} = 0 \\
x &= C\,(-86 - D) = 0.0625\,(-86 - 0) = -5.375
\end{aligned}
$$

Získali jsme tedy stejný výsledek, jako v případě reprezentace s pevnou řádovou čárkou.

Kvantizované vrstvy Jak již bylo zmíněno, tak některé knihovny pro konvoluční neuronové sítě již podporují tuto reprezentaci.

Přesněji se jedná o framework *Tensorflow*, který provádí kvantizaci na osmi bitech [2]. Minimální a maximální hodnota je zjišťována při převodu (je předpočítána maximální a

minimální hodnota na základě minimálních a maximálních hodnot vstupů operací a toho, že prováděné operace jsou známé). Podporovány jsou jen některé vrstvy. Váhy, vstupy a výstupy jsou tedy uloženy na 8 bitech, výpočet je však prováděn na 32 bitech (vzhledem k možnosti přetečení) a výsledek je poté opět převeden na 8 bitů. Definice datového typu pro vstupní a výstupní hodnoty se liší – minimum a maximum intervalu je odlišné.

Podporu lze také nalézt ve frameworku *Caffe*. Přesněji se jedná o nástroj *Ristretto* [16], který podporuje různý počet bitů a více formátů. Interně je však výpočet stále prováděn v datových typech s plovoucí řádovou čárkou. Kvantizovány jsou pouze váhy a vstupy/výstupy vrstev.

Výhody knihovny a `FixedPoint` Výhodou přístupu použitém pro tuto knihovnu je, že uživatel bude mít větší kontrolu nad datovými typy a jejich šířkou – bude moci využívat až tři různé, což například řešení *Tensorflow* neumožňuje. Stejně tak veškeré výpočty budou prováděny v této reprezentaci, zatímco v případě *Tensorflow* vyžaduje například násobení matic akumulátory s větší bitovou šířkou a *Ristretto* výpočty interně provádí v datovém typu s plovoucí řádovou čárkou.

Uživatel dále bude schopen definovat i vlastní datové typy (lze tedy bez problémů přidat i kvantizaci) a také definovat vlastní operace – což třeba dále umožní ověřovat vliv aproximovaných funkcí.

V neposlední řadě je také třeba zmínit, že operace nad reprezentací s pevnou řádovou čárkou jsou jednodušší a mají menší režii než operace nad kvantizovanou reprezentací.

Nevýhody `FixedPoint` Velkou předností kvantizace a jedním z důvodů, proč se používá, je schopnost vyjádřit prakticky libovolně velký interval (samozřejmě ale platí, že čím větší rozsah, tím menší přesnost). Při výpočtech tedy nemůže dojít k přetečení, což je velkým problémem v případě reprezentace s pevnou řádovou čárkou jak bude ukázáno dále.

Schopnost vyjádřit libovolný interval má také velmi příznivý vliv na přesnost sítě. Použití kvantizace tedy prakticky ve všech případech bude produkovat lepší výsledky – avšak jak již bylo zmíněno, kvantizace má vyšší režii než jednodušší provádění operací v pevné řádové čárce (kdy reprezentace je totožná napříč všemi vrstvami).

4.4 Testování

Součástí zdrojových kódů knihovny by měly být i *unit testy* (například pomocí frameworku *Google Test*[1]). Testy mají pozitivní vliv na vnímání knihovny potencionálním uživatelem, ale hlavně přispívají k testování správnosti algoritmů a odhalení regresí během vývoje.

Testována bude jak správnost pomocných funkcí knihovny a dalších algoritmů, tak i správnost implementace operací vrstev konvoluční neuronové sítě. To bude prováděno na základě předpočítání očekávaných hodnot a jejich porovnání s výstupy vrstev (jak dopředného, tak zpětného průchodu).

Mimo to bude knihovna testována na úlohách typických pro konvoluční neuronové sítě (budou ověřovány různé kombinace vrstev, aktivačních funkcí, optimalizátorů, ztrátových funkcí atd.) s ohledem na schopnost učení.

Stejně tak bude knihovna porovnána s jinými volně dostupnými na internetu, aby bylo ověřeno, že kvalita učení je srovnatelná a nezaostává za ostatními. Výsledky těchto testů jsou dostupné v kapitole 7.

[1] Dostupné z https://github.com/google/googletest.

Během vývoje bude využíván i regresní test napsaný v jazyce `Python`, který bude periodicky provádět trénování nad pevně danými kombinacemi hyperparametrů a datových sad. Účelem tohoto testu bude ověřit, že se nezhoršuje kvalita učení nebo časová náročnost trénování. Tento test nebude součástí odevzdání, neboť pro dokončenou knihovnu již nemá své opodstatnění. K odhalení regresí poté slouží jednotkové testy a srovnání s výstupy trénování, které součástí odevzdání budou.

Kapitola 5

Implementace knihovny

Výsledkem diplomové práce je knihovna pro práci s konvolučními neuronovými sítěmi, která poskytuje uživateli možnost vytvořit vlastní architekturu, natrénovat ji a následně využívat. I když hlavní využití spočívá v integraci knihovny do svého vlastního projektu, je k dispozici i konzolové rozhraní.

V této kapitole je popsán průběh implementace, především zajímavé implementační detaily a problémy, které bylo třeba vyřešit.

5.1 Implementační jazyk a použité knihovny

Knihovna je naimplementována v jazyce `C++` za použití standardu `C++14`. Tento jazyk byl zvolen, jelikož trénování konvolučních neuronových sítí je výpočetně velmi náročnou disciplínou a rychlost je velmi podstatným faktorem vypovídajícím o kvalitě knihovny. Programy vytvořené v jazyce `C++` jsou typicky velmi rychlé a velmi často je tento jazyk použit také pro implementaci konvolučních neuronových sítí (i když například rozhraní pro práci s nimi je pak v jiném jazyce).

I když jsou všechny podstatné části knihovny implementovány svépomocí, bylo využito několika open-source knihoven na *pomocné práce*. Tyto knihovny jsou součástí zdrojových kódů (s příslušnými licencemi) a zde budou zmíněny.

TinyXML2 Knihovna pro práci s XML soubory – jejich čtení a vytváření. V knihovně se používá pro načítání a ukládání architektury konvoluční neuronové sítě.

CxxOpts Vzhledem k velkému množství argumentů konzolového rozhraní byla využita knihovna pro zpracování argumentů příkazové řádky (neboť takový subsystém není součástí jazyka `C++`).

LodePNG Pro účely načítání vstupních a trénovacích dat je využita knihovna *LodePNG*, která umožňuje práci se soubory typu `*.png` (*Portable Network Graphics*). Mimo to je knihovna také využita pro ukládání instancí třídy `Image` do souboru, díky tomu si lze například zobrazit načtené vstupy složitějších formátů nebo také filtry konvoluční vrstvy či vstupy a výstupy jednotlivých vrstev sítě.

5.2 Konvoluční neuronová síť

Implementace konvoluční neuronové sítě sleduje třídní diagram (viz obrázek 4.1). V této kapitole budou zmíněny pouze zajímavé implementační detaily a poté snahy o zrychlení trénování.

Čitelnost kódu Jedním z problémů, který byl pozorován u mnoha větších knihoven dostupných na internetu, byla špatná čitelnost zdrojových kódů. To je do jisté míry pochopitelné, neboť vývojáři cílí na maximální možnou efektivitu. V případě menších projektů se tomuto problému lze lépe vyvarovat. V této implementaci tedy byla snaha o skloubení rychlosti a čitelnosti kódu (dáno jak jeho strukturou, tak komentáři) – tak aby zdrojové kódy bylo například možno využít i k výukovým účelům.

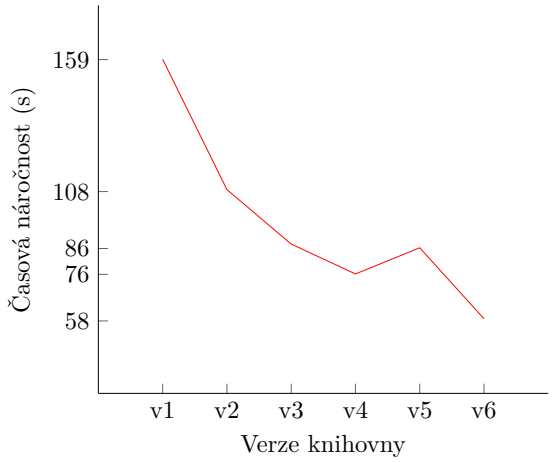
Posouzení úspěšnosti splnění tohoto bodu je ponecháno na čtenáři, neboť tento aspekt je značně subjektivní. Během implementace však byly komentovány hlavičky všech funkcí a v případě, že se vyskytují na rozhraní třídy i jednotlivé parametry a návratové hodnoty. Komentáře v kódu jsou také časté, zejména u kusů kódu, u kterých na první pohled nemusí být patrné, jak fungují. Některé moduly byly přepisovány i jen proto, aby se zvýšila jejich čitelnost (například pokud bylo využíváno duplicitního kódu atd.).

Uložení v paměti K uložení dat a vah je využíváno klasických polí známých z jazyka `C`. Využití vektorů totiž zpomalovalo algoritmy a navíc bylo paměťově náročnější (což v případě, že se pokoušíme načíst například 60 000 trénovacích obrázků, může být problém). Tento linearizovaný formát matice je také v maximální možné míře využíván algoritmy, pro které není rozhodující struktura matice (aktivační vrstvy, dropout vrstvy, úprava matic vah atd.). Aby nedocházelo k problémům s neuvolněnou pamětí je v celé implementaci využito chytrých ukazatelů, *smart pointers*, které jsou v `C++` dostupné od standardu `C++11`.

Urychlení algoritmů Pro urychlení konvolučních a poolingových vrstev byla zvolena optimalizace ve formě předpočítání hran vedoucích k výstupním neuronům během inicializace vrstvy – metoda `createEdges()`. Dopředný a zpětný průchod pak využívá pouze tyto hrany a nedochází tedy k posouvání „okna“ po vstupní matici. Zrychlení je značné, neboť odpadají veškeré výpočty pro určení posuvů, hlídání okrajů nebo doplnění o *zero padding* v případě konvoluční vrstvy.

Mimo těchto urychlení byl po celou dobu implementace kladen důraz na maximální možnou efektivitu algoritmů. Některé části kódu byly i několikanásobně přepisovány pro dosažení co největší přehlednosti a rychlosti.

Efektivita implementace z pohledu časové náročnosti byla průběžně vyhodnocována a na obrázku 5.1 lze vidět její vývoj. Předmětem experimentu bylo vždy změření doby, kterou zabere jedna epocha trénování a následná validace na datové sadě *MNIST* – a to v případě jednoduché architektury s jednou konvoluční, jednou poolingovou a jednou plně propojenou vrstvou. První verzí byl počáteční prototyp schopný trénování, v dalších iteracích pak probíhalo refaktorování kódu, oprava chyb a přidávání nových funkcí. Díky zmíněnému refaktoru se dařilo i při nové funkcionalitě snižovat časovou náročnost. V jedné z pozdějších fází, *v5*, pak byly do knihovny přidány další optimalizátory, chybové funkce a šablony (kvůli nezávislosti na datovém typu), což mělo negativní dopad na rychlost. Proto poté bylo opět nutno refaktorovat kód, tentokrát i s využitím *profileru* dostupném ve *Visual Studio 2017* (jedna z výhod toho, že knihovna je multiplatformní), a časovou náročnost se opět podařilo snížit.



Obrázek 5.1: Vývoj časové náročnosti implementace v případě jedné epochy trénování a následné validace na jednoduché architektuře (stejná architektura i nastavení pro jednotlivé verze)

5.3 Podpůrné prostředky

Jádrem knihovny je samozřejmě konvoluční neuronová síť, tedy implementace různých vrstev, trénování atd. Aby ale bylo možno knihovnu efektivně využívat, poskytuje i množství dalších prostředků. Jejich implementační detaily budou popsány v následujících odstavcích.

5.3.1 Perzistence

Architekturu konvoluční neuronové sítě je možno popsat jednoduchým XML souborem, k jehož načítání a ukládání slouží externí knihovna *TinyXML2*. Vzhledem k nutnosti převodu enumeračních tříd (`enum class`) do řetězců byl přidán modul `PersistenceMapper`, který tyto převody provádí. Své využití také nalezne v případě konzolového rozhraní (nutný opačný převod z textového argumentu příkazové řádky na hodnotu enumerační třídy).

5.3.2 Načítání vstupních dat

Implementováno je načítání tří možných formátů vstupních dat, které jsou blíže popsány v následujících odstavcích.

IDX soubory IDX soubory jsou využívány pro ukládání multidimenzionálních dat. Tento formát se typicky využívá pro tzv. *benchmark* úlohy, jako je například databáze *MNIST* [25].

Formát IDX kromě samotných dat obsahuje i metadata popisující jejich strukturu – šířku, výšku, počet kanálů a celkový počet obrázků.

Parser v tomto případě umožňuje pouze načítání trénovací sady, nikoliv jednoho vzorku.

V případě konzolového rozhraní se předpokládá, že soubor s očekávanými výstupními daty má stejný název jako soubor se vstupními daty, avšak slovo *images* je nahrazeno slovem *labels* (neboť jako parametr se předává pouze cesta k jednomu souboru).

Binární soubory Binární soubory jsou také využívány pro *benchmark* úlohy, například v databázi *CIFAR-10* [22]. V tomto případě jsou v souborech uložena pouze data a to po řádcích. Před každým obrázkem je v jednom bytu zapsaná očekávaná třída.

Uživatel tedy musí zadat rozměry vstupních obrázků. Na základě těchto rozměrů je pak vypočítán počet obrázků, podle velikosti vstupního souboru, a všechny jsou načteny.

Parser v tomto případě umožňuje pouze načítání trénovací sady, nikoliv jednoho vzorku.

Png soubory Uživatelé často své CNN trénují na obrázcích, které sami získali. Pro tyto účely je k dispozici i načítání buď jednotlivých obrázků (při využívání natrénované sítě) nebo celých adresářů (pro účely trénování a validace).

Pro inferenci je k dispozici funkce `parseInputImage()`, která načte obrázek ze zadané cesty do interního formátu využívaného konvoluční neuronovou sítí.

Pro trénování a validaci je k dispozici funkce `parseLabelledImages()` načítající celý adresář. V tomto případě se zadává cesta k textovému souboru, který na každém řádku vždy obsahuje cestu k obrázku (relativně k tomuto souboru) a poté mezerou oddělené očekávané výstupní hodnoty. V případě regresních úloh se jedná přímo o očekávané výstupy, v případě klasifikačních úloh je vhodné zadat 1 pro očekávanou výstupní třídu a 0 pro ostatní třídy (počet výstupních hodnot je stejný, jako počet tříd).

5.3.3 Konzolové rozhraní

Jak již bylo zmíněno, knihovnu je možno využívat jak z jazyka `C++` ve vlastním projektu (viz příklady použití v kapitole 6), ale existuje i konzolové rozhraní. Knihovnu je tak možno využít prakticky z libovolného jazyka a konzolové rozhraní také významně usnadňuje provádění experimentů – není nutno provádět změny v kódu (který musí být následně přeložen, což vzhledem k použití šablon může trvat delší dobu).

V rámci konzolového rozhraní je možno provést následující:

- načíst architekturu sítě (včetně již naučených parametrů) ze souboru typu `XML`,
- provést inferenci nad jedním vstupním souborem typu `PNG`,
- provést validaci nad binárními daty, daty typu `IDX` a adresářem se soubory `PNG`,
- specifikovat posunutí a počet vzorků pro validaci,
- provést trénování (včetně uložení naučených parametrů) nad binárními daty, daty typu `IDX` a adresářem se soubory `PNG`,
- specifikovat posunutí a počet vzorků pro trénování,
- specifikovat ztrátovou funkci,
- specifikovat optimalizátor provádějící úpravu učitelných parametrů,
- nastavit parametry trénování (počet epoch, učící koeficient, velikost *batch*, ...),
- nastavit *seed* pro generátor náhodných čísel (za účelem reprodukování experimentů),
- specifikovat zda má být po každé epoše provedena validace (s tím pak souvisí možnost ukládání pouze vah s nejlepším dosaženým výsledkem na validační sadě),

- specifikovat zda mají být trénovací data náhodně „zamíchána“ před začátkem epochy.

I když konzolové rozhraní nabízí široké množství přepínačů, tak není možné, aby poskytovalo veškerou funkcionalitu, která je jinak dostupná z jazyka `C++`. Jedná se o následující omezení:

- v případě optimalizátorů lze ovlivnit pouze učící koeficient η a úbytek vah μ, žádné další parametry,
- je možno využít pouze předdefinovaných parserů vstupních dat (je navíc nutno dodržet formát dat a příponu souborů),
- není možno provádět další akce spojené s analýzou sítě – například uložení filtrů do souborů typu `PNG`,
- není možno ovlivňovat některé další parametry spojené s trénováním (například průběžné změny učícího koeficientu η v závislosti na epoše).

5.4 Typ s pevnou řádovou čárkou

Jak již bylo řečeno v kapitole 4.3, tak typ s pevnou řádovou čárkou je založen na datovém typu `int32_t`, tedy celočíselném datovém typu se šířkou 32 bitů. Implementace byla založena na ukázkovém kódu *Khurama Aliho* [3]. Tento typ v původním znění umožňuje specifikovat pouze počet bitů za řádovou čárkou a celá část poté využívá zbylé bity. Navíc bylo při provádění testů zjištěno, že řada operací nefunguje správně. Bylo tedy nutno provést zásadní změny.

Typ využitý v této knihovně nese název `FixedPoint` a jedná se o šablonu, která jako parametr vyžaduje dvě čísla – první z nich, F, specifikuje počet bitů před řádovou čárkou a druhé, P, počet bitů za řádovou čárkou. Použití je tedy následující – `FixedPoint<F, P>`. Kromě toho bylo potřeba dodefinovat (a případně opravit) všechny potřebné operace, a to jak aritmetické, tak například operace pro výpis typu atd. To, že všechny operace jsou prováděny správně, bylo důkladně testováno jednotkovými testy.

Aby bylo možno omezit i celou část bylo nutno dodefinovat dvě bitové masky (které se staticky počítají dle parametrů šablony), jedna reprezentuje minimální vyjádřitelné číslo a druhá maximální. V případě, že během provádění určité aritmetické operace (anebo při inicializaci) je jedna z těchto hodnot překročena, tak je hodnota nastavena na maximální, respektive minimální. Tomu říkáme *datový typ se saturací*.

Rozsáhlejší úpravy si ale vyžádala podpora libovolného typu v rámci celé konvoluční neuronové sítě a dalších částí knihovny. Velká část kódu je obalena šablonami. V rámci funkcí je pak třeba důsledně dbát na přetypování – nejen proto, aby se všechny operace opravdu prováděly dle definované implementace, ale i proto, aby knihovnu vůbec bylo možno přeložit (a to bez varování překladače, kdy úroveň varování byla zvolena tak, aby nic neopomíjela).

Testovány byly různé kombinace datových typů `float`, `double` a `FixedPoint<F, P>`. V případě jednoduché sítě ale bylo možno provádět trénování například i na typu `int`, avšak samozřejmě se špatnými výsledky.

V rámci experimentálního příkladu byl také vytvořen ukázkový program, který umožňuje, aby různé vrstvy sítě měly různé datové typy (například s různým počtem bitů před a za řádovou čárkou) pro dopředné provádění a váhy, což si také vyžádalo nemalé zásahy

do kódu. Kvůli tomu musela být také přidána speciální vrstva `ConversionLayer`, která provádí převedení datových typů mezi dvěma vrstvami, jejichž datový typ pro inferenci se liší (jedna vrstva má například typ `FixedPoint<8,8>` a druhá `FixedPoint<4,4>`).

Omezení Kvůli použití celočíselného typu, jehož rozsah je relativně malý (oproti datovým typům s plovoucí řádovou čárkou), může dojít k přetečení proměnné celočíselného typu, která je schována uvnitř datového typu `FixedPoint`.

To se může stát například při násobení dvou velkých kladných čísel. Uvažujme datový typ `FixedPoint<12,12>` a v něm vyjádřené číslo 12. Celé číslo, které mu odpovídá v této reprezentaci je 49 152. Pokud však vynásobíme dvě takto velká čísla, dostaneme hodnotu, která je vyšší, než je možno vyjádřit na celočíselném typu s 32 bity, dojde tedy k přetečení a výsledek násobení dvou kladných čísel je záporný.

Aby bylo jisté, že k tomu nedojde, je možno využít celkem pouze 16 bitů. Dále také musí platit, že počet bitů před řádovou čárkou musí být větší než 0 (abychom vyjádřili alespoň znaménko). Proto jsou doporučena tato omezení:

$$F > 0 \tag{5.1}$$

$$0 < F + P \leq 16 \tag{5.2}$$

Lze využívat i větší šířku, ale je třeba počítat s tím, že se tento problém může vyskytnout. To je samozřejmě omezující, a proto knihovna dále obsahuje typ `FixedPoint64<F,P>`, který vnitřně využívá typu `int64_t`. Vzhledem k tomu, že pro většinu experimentů bude 16 bitů postačujících, tak je v knihovně zachována jak 32, tak 64, bitová, varianta. Pro tento typ tedy platí následující omezení:

$$F > 0 \tag{5.3}$$

$$0 < F + P \leq 32 \tag{5.4}$$

Kapitola 6

Použití

Součástí této kapitoly je popis použití implementované knihovny, funkcionality, kterou nabízí, a dalších důležitých informací, které by měl potencionální uživatel znát.

V případě, že si uživatel přeje využívat programové rozhraní, tak stačí do svého projektu zahrnout soubor `src/ConvolutionalNeuralNetwork.h` nebo `src/NeuralNetwork.h`. Případně pak také hlavičkové soubory dalších modulů – viz `src/Parsers/` a `src/Utils/`. Je vhodné se také seznámit s třídním diagramem na obrázku 4.1.

Další pokyny k použití knihovny lze nalézt v příloze A.

6.1 Nabízená funkcionalita

Část nabízené funkcionality již byla zmíněna v předchozích kapitolách, na tomto místě provedeme rekapitulaci a doplníme již zmíněné informace o další nabízenou funkcionalitu.

Konvoluční neuronová síť

- Vrstvy
  - Konvoluční vrstva
  - Plně propojená vrstva
  - Drop-out vrstva
  - Poolingová vrstva
    - Operace *max*
    - Operace *average*
  - Aktivační vrstva
    - Sigmoid
    - ReLU
    - Leaky ReLU
    - Tanh
    - Softmax
- Ztrátové funkce
  - Mean squared error

  - Cross entropy (jak pro binární klasifikaci, tak pro klasifikaci do více tříd)

- Optimalizátory
  - SGD
  - Momentum
  - Nestorov momentum
  - Adagrad
  - Adam

Podporované formáty vstupních dat V případě použití programového rozhraní je možno využít libovolný formát, pro který uživatel vytvoří převod do vektoru (v případě rozhraní pro neuronovou síť) nebo do trojrozměrné matice `Image` (v případě rozhraní pro konvoluční neuronovou síť). Třemi formáty, které jsou v základu podporovány, jsou:

- binární data uložená po řádcích,
- formát IDX včetně metadat,
- obrazové soubory typu `*.png`.

Modul perzistence Modul perzistence umožnuje uložení natrénované konvoluční neuronové sítě na disk a její opětovné načtení, a to včetně natrénovaných parametrů. Součástí modulu je také XML formát (viz ukázka 4.3), který popisuje architekturu sítě. Natrénované parametry jsou uloženy v textových souborech.

Výstupní modul Součástí knihovny je také modul `ImageUtils`, který umožňuje pracovat s instancemi typu `Image` za účelem jejich vypsání nebo uložení na disk ve formě souboru typu `*.png`.

6.1.1 Tipy a triky

Během používání knihovny bylo identifikováno několik poznatků, které by měl uživatel znát. Jedná se i obecné informace vztahující se ke konvolučním neuronovým sítím, které nejsou platné jen pro knihovnu *TypeCNN*.

Konvergence V případě trénování konvoluční neuronové sítě je potřeba sledovat průběžný vývoj učení, zda konverguje ke správnému řešení. Konvergenci ovlivňuje velké množství nastavení. Velmi podstatným je vhodná volba optimalizátoru a především jeho parametrů. Optimalizátory mají defaultní hodnoty parametrů nastaveny podle obecných doporučení. Pokud však síť konverguje velmi pomalu, nebo případně učení stagnuje či dokonce diverguje, je třeba zvolit jiné hodnoty. Ty jsou závislé od zvolené ztrátové funkce, zvolených aktivačních funkcí, zvolené architektury a na dalších skutečnostech. V případě divergence je typicky potřeba snížit učící koeficient. V případě pomalé konvergence může být problém v nízkém koeficientu učení. V případě stagnace je pak vhodné provést taktéž snížení koeficientu učení, avšak změna nemusí být tak velká jako v případě divergence. Vhodné je také aktivovat náhodné „zamíchání“ dat, které často vede k lepším výsledkům.

Pro účely sledování průběhu učení je možno zapnout průběžné výstupy trénování, které vypisují chybu v předem definovaných krocích a také po každé epoše. Tyto výpisy jsou

defaultně povolené při využívání konzolového rozhraní, v případě programového rozhraní je možno je aktivovat. Případně je tato chyba součástí parametru *callback* funkce, kterou lze nastavit a volá se vždy na konci epochy.

Vliv má samozřejmě také architektura sítě. Čím rozsáhlejší architektura (a s tím spojený větší počet vah), tím pomalejší konvergenci lze očekávat. Typický vývojem je pomalejší konvergence na počátku, poté rychlejší a ke konci opět pomalejší. Při vhodně zvolených parametrech bude konvergovat libovolná architektura, avšak složitost úlohy a velikost sítě udávají, do jaké míry je trénování schopno dosáhnout požadované úspěšnosti na validační sadě.

Při extrémním případu divergence se může stát, že dojde k přetečení nebo podtečení během provádění. To je automaticky detekováno, trénování ukončeno a uživatel je o této skutečnosti informován. Dochází k tomu při nevhodné volbě parametrů – viz výše.

Přeučení Jak již bylo zmíněno, problém přeučení je charakterizován snižováním chyby trénovací sady, avšak stoupající chybovostí na sadě validační. K detekci tohoto problému je možno zapnout periodickou validaci po každé epoše (snižování validační přesnosti o nízké hodnoty během trénování ještě ale není příznakem přeučení, pokud se tedy nejedná o několik epoch po sobě). Knihovna disponuje dvěma možnostmi prevence přeučení a to:

- drop-out vrstva – přidání drop-out vrstev má typicky menší negativní vliv na rychlost konvergence, avšak fakt, že vstupy jsou náhodně měněny, může mít pozitivní vliv na zabránění přeučení,
- snižování vah – všechny optimalizátory obsahují i parametr *weight decay*, který při každé úpravě vah sníží jejich hodnotu o malou část, lze tak zabránit tomu, aby se váhy a biasy stávaly příliš velkými, což má také pozitivní dopad na problém přeučení.

Cross entropy Ztrátovou funkci *cross entropy* je vzhledem k její definici možno využít pouze v případě, že výstupní vrstva má všechny hodnoty v intervalu $\langle 0, 1 \rangle$. To splňují aktivační vrstvy *Softmax* a *Sigmoid*.

6.2 Konzolové rozhraní

Funkcionalitu poskytovanou konzolovým rozhraním je nejvhodnější popsat argumenty, které nabízí. Ty lze vidět v tabulce 6.1. Obdobnou tabulku lze vypsat v angličtině pomocí přepínače `-h`.

6.3 Programové rozhraní

Programové rozhraní nabízí stejné možnosti jako konzolové rozhraní. Navíc pak umožňuje využívat i dalších funkcí, které v případě využití z konzole nejsou dostupné.

V ukázce 6.1 lze vidět načtení trénovacích a validačních dat pro datovou sadu *MNIST*, poté vytvoření konvoluční neuronové sítě s pěti vrstvami (konvoluční, aktivační, poolingová, plně propojená a opět aktivační vrstva), natrénování této sítě a nakonec její validaci. Jak lze vidět, tak je možno ovlivnit veškerá nastavení trénování. Vzhledem k tomu, že optimalizátor si stále uchovává svůj konkrétní typ (a nikoliv abstraktní typ `IOptimizer` jako v případě konzolového rozhraní), je možno nastavit i další parametry jako třeba *momentum*.

Přepínače		Popis
Obecné argumenty		
`-h`	`--help`	Zobrazí nápovědu.
`-c`	`--cnn`	Cesta k XML souboru popisující konvoluční neuronovou síť.
`-g`	`--grayscale`	Specifikuje, že pracujeme se vstupy s barvami v úrovních šedé (nutno pouze při využití formátu `*.png`).
	`--type-info`	Vypíše informace o typech používaných pro váhy, inferenci a trénování.
Inference		
`-i`	`--input`	Vstupní obrázek (`*.png`) pro inferenci s načtenou sítí.
Validace		
`-v`	`--validate`	Soubory (oddělené mezerou) pro načtení vstupních dat pro validaci, o typu *parseru* se rozhoduje na základě regulárního výrazu (`.+txt.*`, `.+bin.*`, `.+idx*`).
	`--validate-num`	Počet vzorů načítaných k validaci.
	`--validate-offset`	Posunutí ve validačním datasetu.
Trénování		
`-t`	`--train`	Soubory (oddělené mezerou) pro načtení vstupních dat pro trénování, platí to samé jako v případě validační sady.
	`--train-num`	Počet vzorů načtených pro trénování.
	`--train-offset`	Posunutí v trénovacím datasetu.
`-s`	`--seed`	*Seed* pro inicializaci generátoru náhodných čísel (pro reprodukci experimentů).
`-e`	`--epochs`	Počet epoch pro trénování.
`-l`	`--learning-rate`	Učící koeficient η.
`-d`	`--weight-decay`	Koeficient úbytku vah μ.
`-b`	`--batch-size`	Velikost shluku dat, po kterém se provádí úprava vah.
	`--do-not-load`	Specifikuje, že se nemá provádět načítání vah z textových souborů.
	`--do-not-save`	Specifikuje, že se nemá provádět ukládání natrénovaných vah.
	`--optimizer`	Specifikuje použitý optimalizátor. Lze vybrat z `sgd`, `sgdm`, `sgdn`, `adam`, `adagrad`. Defaultním je `sgd`.
	`--loss-function`	Specifikuje použitou ztrátovou funkci. Lze vybrat z `MSE`, `CE` a `CEbin`. Defaultní je `MSE` (střední kvadratická chyba).
	`--periodic-validation`	Specifikuje, že se má provádět validace po každé epoše (jinak jen na konci trénování).
	`--periodic-output`	Specifikuje frekvenci výpisu aktuální chyby během trénování (počet vzorků, po kterých se výpis provede).
	`--shuffle`	Specifikuje, že se má provádět „zamíchání“ trénovacích dat před začátkem každé epochy
	`--keep-best`	Specifikuje, že natrénované parametry se budou ukládat průběžně a pouze pokud bude validační chyba nižší pro aktuální parametry než pro dříve uložené.

Tabulka 6.1: Argumenty konzolového rozhraní

Více příkladů lze pak nalézt ve složce `examples/`, která je součástí knihovny. Stejně tak je vhodné prozkoumat soubor `src/CommandLineInterface.cpp`, který implementuje funkcionalitu konzolového rozhraní a obsahuje příklady použití většiny modulů knihovny.

Názvy vrstev je vhodné přebírat ze souboru `src/LayerAliases.h`, který obsahuje typové aliasy, které obsahují i definici datových typů šablon.

```
auto trainingData = IdxParser::parseLabelledImages("mnist/train-images.idx3",
                                                   "mnist/train-labels.idx1", 10);
auto validationData = IdxParser::parseLabelledImages("mnist/test-images.idx3",
                                                     "mnist/test-labels.idx1", 10);

auto inputDimensions = trainingData[0].first.getDimensions();

auto layer1 = std::make_shared<Convolution>(inputDimensions, 1, 8, 5, 0, true);
auto layer2 = std::make_shared<ReLU>(layer1->getOutputSize());
auto layer3 = std::make_shared<MaxPooling>(layer2->getOutputSize(), 2, 2);
auto layer4 = std::make_shared<FullyConnected>(layer3->getOutputSize(),
                                               Dimensions{ 10, 1, 1 }, true);
auto layer5 = std::make_shared<Sigmoid>(layer4->getOutputSize());

auto cnn = ConvolutionalNeuralNetwork(TaskType::Classification);
cnn.addLayer(layer1);
cnn.addLayer(layer2);
cnn.addLayer(layer3);
cnn.addLayer(layer4);
cnn.addLayer(layer5);

TrainingSettings settings;
settings.epochs = 5;
settings.errorOutputRate = 10000;
settings.shuffle = true;

cnn.enableOutput();

auto optimizer = std::make_shared<SgdWithMomentum>();
optimizer->momentum = 0.6f;

cnn.train(settings, trainingData, LossFunctionType::MeanSquaredError, optimizer);

cnn.validate(validationData);
```

Ukázka 6.1: Ukázkové použití programového rozhraní

Dalším důležitým rozšířením, které by měl uživatel znát, je možnost nastavení *callback* funkce pomocí metody `setOnEpochFinishedCallback()` rozhraní konvoluční neuronové sítě. Tato metoda se volá vždy po ukončení epochy a uživateli poskytuje informace o aktuální epoše, aktuální chybě, validační úspěšnosti (pokud byla zapnuta periodická validace), ale i přístup ke struktuře `TrainingSettings`. Uživatel tak může měnit nastavení trénování za běhu – což se hodí zejména pro změnu učícího koeficientu v závislosti na epoše.

Taktéž je možno využít i modulu `ImageUtils`, který je z konzolového rozhraní nepřístupný. Stejně tak je možno použít i rozhraní `NeuralNetwork` pro dopředné plně propojené

neuronové sítě. Jedná se o nadstavbu nad rozhraním pro konvoluční neuronové sítě, které umožňuje specifikovat sekvenci plně propojených vrstev následovaných aktivačními vrstvami. Namísto matic jsou vstupem inference, trénování a validace jednorozměrné vektory.

6.4 Změna datového typu

V základním nastavení knihovna podporuje nastavení různých datových typů pro inferenci – `ForwardType`, trénování – `BackwardType` a váhy – `WeightType`. Typy je možno specifikovat v souboru `Makefile` nebo `src/CompileSettings.h`, pokud je knihovna využívána ve vlastním projektu.

Podporované a testované jsou libovolné kombinace typů `double`, `float`, `FixedPoint<F, P>` a `FixedPoint64<F, P>`, kde F odpovídá počtu bitů před řádovou čárkou a P počtu bitů za řádovou čárkou.

Vzhledem k využití šablon je možné používat různé datové typy mezi jednotlivými vrstvami. V tomto případě však není možno využít ani konzolové, ani programové rozhraní. Příklad takového přístupu je k dispozici v souboru `examples/fixed_point.cpp`.

V případě, že by chtěl uživatel použít vlastní implementace operací nad datovým typem s pevnou řádovou čárkou (například aproximované operace), je možno upravit definici datového typu v souboru `src/Utils/FixedPoint.h`.

V případě, že by uživatel chtěl dodefinovat vlastní datový typ, je nutné implementovat minimálně operace, které lze nalézt v témže souboru. To zahrnuje aritmetické operace, logické operace, implicitní konverzi datového typu do typu `float`, operaci pro využití datového typu ve výpisech typu `std::ostream`. Dále je nutné poskytnout i metody `getMinimumValue()`, `getMaximumValue()` a `getEpsilonValue()`, ty popořadě vrací nejnižší vyjádřitelnou hodnotu, nejvyšší vyjádřitelnou hodnotu a rozdíl mezi dvěma po sobě jdoucími hodnotami.

6.5 Příklady použití

Součástí knihovny jsou také ukázky použití knihovny ve složce `examples/`. Ty budou v krátkosti popsány v následujících odstavcích.

`benchmark.cpp` Implementace použití knihovny s architekturou konvoluční neuronové sítě a nastaveními, které byly použity pro srovnání knihoven.

`neural_network.cpp` Ukázka použití rozhraní `NeuralNetwork` určeného pro obyčejné neuronové sítě.

`demo.cpp` Ukázka načtení již naučené sítě a jejího použití. Po načtení konvoluční neuronové sítě natrénované na datové sadě *MNIST* je periodicky načítán vstupní `*.png` soubor. Při každé jeho změně je spuštěna inference a na standardní výstup vypsán její výsledek. Je možno využít přiložené natrénované sítě a soubor `sample.png`, který má validní rozměry pro datovou sadu *MNIST* a lze do něj malovat bílou barvou například v programu *Malování*.

`fixed_point.cpp` Ukázka použití různých datových typů mezi jednotlivými vrstvami konvoluční neuronové sítě (jak již bylo zmíněno, tak není možno použít klasické programové rozhraní). Tato ukázka lze použít jako základ pro experimenty s tímto charakterem.

6.6 Natrénované konvoluční neuronové sítě

Dále lze také v repozitáři nalézt příklady natrénovaných konvolučních neuronových sítí na datových sadách *MNIST* a *CIFAR-10* a to v adresáři `results/`. Součástí jsou vždy následující soubory:

- `*.xml` – soubor popisující architekturu sítě,
- `*.txt` – soubory s natrénovanými vahami,
- `*.sh` – shell skript použitý pro trénování,
- `*.log` – výstup konzolového rozhraní během trénování.

6.7 Datové sady

Ve složce `resources/` lze nalézt datové sady *MNIST* a *CIFAR-10*. První z nich obsahuje čtyři soubory – trénovací vstupy s očekávanými výstupy a validační vstupy s očekávanými výstupy. Druhá z nich pak obsahuje souborů více, neboť trénovací sada je rozdělena na více souborů (vzhledem k velikostním limitům *GitHub*). Ty je možno předem spojit příkazem:

```
Linux:     cat data_batch_1.bin ... data_batch_5.bin > data_batch.bin
Windows:   copy data_batch_1.bin + ... + data_batch_5.bin data_batch.bin /B
```

Dále je také k dispozici ukázka adresáře se soubory typu `*.png`, stejný formát musí dodržovat i adresář vytvořený uživatelem. Obsahuje po 20 souborech z datových sad *MNIST* a *CIFAR-10* (vytvořeno pomocí modulu `ImageUtils`) i s popisnými soubory, které se načítají do knihovny.

6.8 Testy

Součástí knihovny jsou také jednotkové testy (anglicky *unit tests*), které ověřují základní funkcionalitu knihovny. Testy lze nalézt v adresáři `tests/`.

Obsaženy jsou především testy jednotlivých vrstev konvoluční neuronové sítě, ale i testy datového typu s pevnou řádovou čárkou (že jsou hodnoty reprezentovány správně a operace poskytují očekávané výstupy).

Pokyny ke spuštění testů lze nalézt v příloze A.

Kapitola 7

Zhodnocení výsledků

V následujících odstavcích bude vyhodnocena implementace knihovny na sérii testovacích scénářů. Vliv datového typu na schopnost sítě učit se bude také podroben experimentům.

Veškeré experimenty byly prováděny na notebooku `Acer Aspire V17 Nitro` vybaveném procesorem `Intel Core i7-4720HQ (2.60 GHz×8)`, 8 GB paměti RAM a operačním systémem `Ubuntu 14.04 LTS (64-bit)`.

7.1 Porovnání s jinými knihovnami

V průběhu implementace byla knihovna porovnávána s předchozími verzemi, aby bylo garantováno, že prováděné změny vedou na rychlejší a kvalitnější učení. To bylo možné díky již zmíněnému skriptu pro regresní testování. Finální implementace pak byla porovnána s dalšími knihovnami volně dostupnými na internetu. V odstavcích níže bude každá z těchto knihoven v rychlosti popsána. Byly vybrány jak malé projekty, tak knihovny, které jsou běžně používané v praxi.

Simple CNN Knihovna *SimpleCNN* [11] je jednoduchá knihovna jednoho autora, která poskytuje jednu aktivační funkci, jednu ztrátovou funkci a jeden optimalizátor. Chybí pomocné moduly jako například perzistence.

Zdrojové kódy jsou psány velmi jednoduše a jsou dobře čitelné. Pro účely srovnání bylo nutné doimplementovat funkci provádějící testování na validační sadě a měření času trénování.

TinyDNN Knihovna *TinyDNN* [31] poskytuje širokou funkcionalitu a jedná se o open-source projekt, na kterém spolupracuje velké množství lidí. Poskytuje mnoho možností urychlení (všechny dostupné na testovacím počítači byly zapnuté), s čímž ale souvisí delší a složitější instalace.

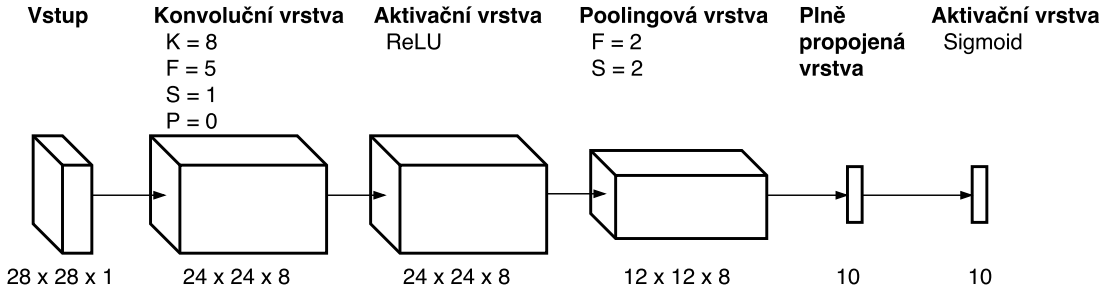
Keras + TensorFlow Obě doposud zmíněné knihovny mají dostupná rozhraní z jazyka `C++`. Knihovna *TensorFlow* [1] má rozhraní pro jazyk `Python`, ale velká část je psána v jazyce `C++`. Vzhledem ke složitosti tohoto rozhraní je pak knihovna často používána ve spojení s knihovnou *Keras* [7], jejíž rozhraní je mnohem jednodušší. Obě jsou knihovny často využívané v praxi. Tensorflow také jako jediná z porovnávaných poskytuje možnost provádění na grafickém procesoru. Poskytuje také vůbec největší množství funkcionality. Knihovnu je

doporučeno nainstalovat se všemi možnými zrychleními, to však vyžaduje kompilaci zdrojových kódů na vlastním počítači. Což vždy nemusí být jednoduché a je časově náročné.

7.1.1 Porovnání

Všechny knihovny byly porovnávány na datasetu *MNIST*. Ten obsahuje 60 000 obrázků pro trénování a 10 000 pro následnou validaci. Architektura, i s nastavením trénovacích konstant, byla převzata z jednoho z příkladů dostupných ke knihovně *SimpleCNN*. Ten byl zvolen proto, že je jednoduchý a tedy použité vrstvy a parametry učení lze nalézt v téměř každé knihovně – proto je vhodný pro srovnání.

Architekturu lze vidět na obrázku 7.1.



Obrázek 7.1: Architektura pro porovnání knihoven

Parametry trénování byly následující:

- optimalizátor gradientního sestupu s využitím momenta,
- stochastický gradientní sestup (velikost dávky 1) – knihovna *SimpleCNN* jinou možnost neposkytuje,
- ztrátová funkce střední kvadratické chyby (*mean squared error*),
- koeficient učení 0.01,
- momentum 0.9,
- úbytek vah 0.0.

Každý experiment byl prováděn pětkrát a výsledná doba trénování a úspěšnost je průměrem ze všech pěti běhů. Konvoluční neuronová síť byla v každé knihovně trénována po dobu deseti epoch.

Název knihovny	**Doba trénování** [s]	**Úspěšnost** [%]
SimpleCNN	746	97.20
TinyDNN	151	98.23
Keras (TensorFlow)	744	98.34
TypeCNN	538	98.31

Obrázek 7.2: Porovnání čtyř knihoven při trénování konvoluční neuronové sítě po dobu 10 epoch (průměr z pěti běhů)

7.1.2 Výsledek

Cílem provedených experimentů bylo ukázat, že vytvořená knihovna je tzv. „konkurence schopná“. Výsledky (viz tabulka 7.2) budou shrnuty v odstavcích níže.

Časová náročnost Z pohledu časové náročnosti knihovna *TypeCNN* překonala knihovnu *SimpleCNN*, která by se co do velikosti dala zařadit do stejné kategorie (projekt jednoho člověka). Nejrychlejší knihovnou pak byla knihovna *TinyDNN* a to s velkým přehledem, byla samozřejmě přeložena se všemi možnými urychleními (speciální instrukce, paralelismus atd.).

Keras s *Tensorflow* je velmi často používán vzhledem k výhodám knihovny *Tensorflow* a jednoduchému rozhraní *Keras*. Existují však případy, kdy je toto spojení pomalejší než využití pouze *Tensorflow*. Vzhledem k tomuto faktu a tomu, že knihovna *Tensorflow* také disponuje daleko rychlejší implementací pro grafické procesory, je potřeba tento výsledek brát s rezervou.

Je také nutno dodat, že použitím *mini-batch* gradientního sestupu místo *stochastického* získáme z časového hlediska nižší hodnoty, kdy zrychlení u *TinyDNN* a *TensorFlow* je mnohem vyšší než v případě knihovny *TypeCNN*, která není paralelizována. Na druhou stranu je ale také potřeba zmínit, že použití větší dávky může vést k pomalejší konvergenci.

Kvalita učení Kvalita učení v případě knihoven *TypeCNN*, *TinyDNN* a *Keras + Tensorflow* byla prakticky totožná. V případě *SimpleCNN* knihovny byl výsledek mnohem horší (v případě použité datové sady a architektury lze méně než 98% považovat za nízký výsledek).

Poskytovaná funkcionalita Z pohledu funkcionality se na předních příčkách umístily knihovny *Keras + TensorFlow* a *TinyDNN*, které jsou ve vývoji delší dobu a podílí se na nich mnoho lidí.

Z pohledu projektů „jednoho člověka“ disponuje vyvinutá knihovna daleko širší funkcionalitou než *SimpleCNN*, neboť nabízí více vrstev a jejich parametrů, optimalizátorů, chybových funkcí, ale také další moduly užitečné pro uživatele (např. modul perzistence, modul načítání dat, konzolové rozhraní a další).

7.2 Případové studie

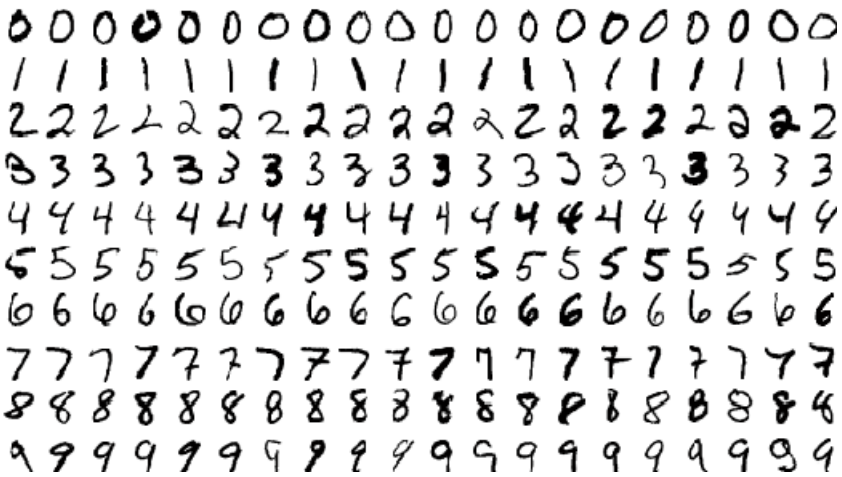
Pro testování konvolučních neuronových sítí existují typické *datasety*, které se používají jako benchmarky pro vyhodnocování úspěšnosti nových vrstev či architektur. Ty budou popsány v této kapitole, společně s tím budou také uvedeny nejlepší výsledky a výsledky dosažené při použití knihovny vytvořené v rámci této práce.

Problémem však je, že těch nejlepších výsledků bylo dosaženo na velmi rozsáhlých architekturách (jejichž trénování by ve vytvořené knihovně trvalo velmi dlouhou dobu) a často také s dalšími rozšířeními, kterými knihovna nedisponuje.

Sítě jsou většinou trénovány pomocí knihoven, na kterých se podílí desítky až stovky lidí. Příkladem může být *Caffe* [20] nebo *TensorFlow* [1]. Nejenže jsou implementovány s ohledem na maximální možnou rychlost, ale jsou také masivně paralelizovány, využívají speciálních instrukcí a typicky i běží na grafických procesorech. Doba trénování je pak nesrovnatelně kratší – to umožňuje využití právě oněch rozsáhlejších architektur, což má pozitivní vliv na přesnost natrénované sítě.

7.2.1 MNIST

Databáze *MNIST* [25] obsahuje 60 000 trénovacích dvojic a 10 000 validačních dvojic s ručně psanými číslicemi v rozsahu $0-9$. Velikost každého obrázku je 28×28 pixelů, hloubka je 1, neboť obrázky jsou v úrovních šedé.



Obrázek 7.3: Ukázka MNIST datasetu [10]

MNIST byla vytvořena z rozsáhlejší databáze *NIST*, odkud byla vybrána jen část obrázků, jejichž velikost navíc byla normalizována a pozice číslic vycentrována.

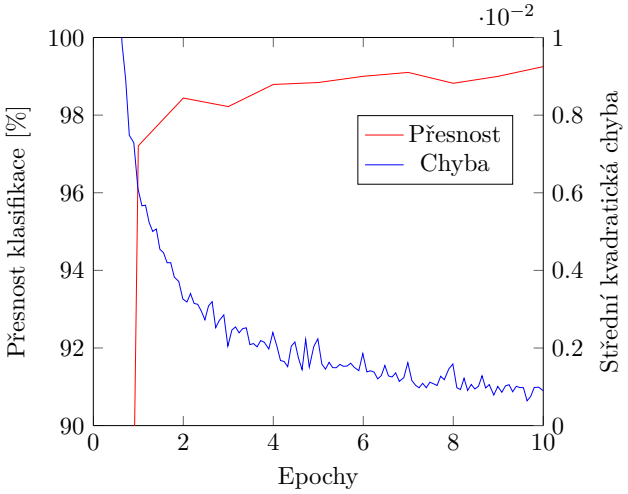
Databáze je k dispozici ve formátu `IDX`, který byl popsán v kapitole 5.3.2. Celková velikost je okolo 55 MB. Ukázku datasetu lze vidět na obrázku 7.3 a je možno si povšimnout, že s rozpoznáním některých čísel by mohl mít problém i člověk. Přesto však aktuálně nejlepší klasifikátory dosahují úspěšnosti 99.7% a vyšší [4].

V rámci diplomové práce byl využíván primárně tento dataset a to pro validaci změn prováděných během implementace. Nejvyšší dosažená úspěšnost byla 99.37%. Tohoto výsledku bylo dosaženo na upravené architektuře *LeNet-5*, jejíž specifikace je součástí odevzdání. Další detaily lze nalézt v kapitole 7.3.3. Tento výsledek již lze považovat za demonstraci schopnosti provádět trénování konvoluční neuronové sítě nejen na tomto datasetu, ale i na dalších úlohách.

Průběh jednoho z trénování lze vidět na obrázku 7.4. Průběh dalších trénování (společně s nastavením trénování) lze pak nalézt v `*.log` souborech přiložených ke knihovně. Vyhodnocování chyby bylo prováděno průběžně během každé epochy, validace pak byla provedena vždy na konci epochy. Lze si povšimnout, že chyba postupem času stále klesá a validační úspěšnost stoupá – avšak s určitými výkyvy.

7.2.2 CIFAR-10

Databáze *CIFAR-10* [22] obsahuje 50 000 trénovacích dvojic a 10 000 validačních dvojic barevných obrázků s 10 tématy. Postupně to jsou – letadla, automobily, ptáci, kočky, jeleni, psi, žáby, koně, lodě a nákladní vozidla. Velikost každého obrázku je 32×32 pixelů a hloubka je 3 – obrázky jsou barevné (`RGB`).



Obrázek 7.4: Ukázka průběhu učení na datové sadě MNIST

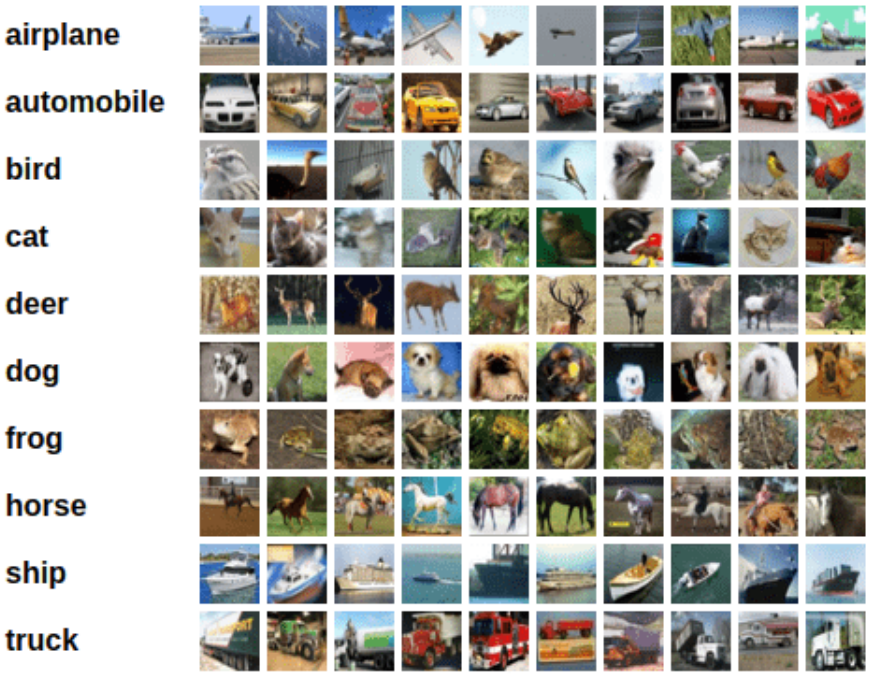
CIFAR-10 byla vytvořena z rozsáhlejší databáze 80 000 000 obrázků. Stejně tak existuje i *CIFAR-100*, která obsahuje stejný počet obrázků, avšak tříd je 100 (ty jsou sdruženy pod několik nadtříd, kterých je 20).

Databáze je k dispozici ve více formátech (například pro *Python* nebo *Matlab*), v této práci ale využíváme binární variantu, která má velikost 184 MB. Ukázku datasetu lze vidět na obrázku 7.5. Zde opět platí, že s rozpoznáním některých obrázků by mohl mít problém i člověk. Pro strojové zpracování je tento dataset mnohem obtížnější než MNIST, zejména kvůli obrovskému rozdílu mezi obrázky stejných tříd (různé natočení, různé pozadí atd.). Jednoduchá neuronová síť je schopna po chvíli trénování dosáhnout úspěšnosti na datové sadě MNIST převyšující 95%, stejná síť a za stejnou dobu trénování však v případě CIFAR-10 dosáhne úspěšnosti pouze okolo 40%. I přesto ale nejlepší klasifikátory dosahují úspěšnosti 96.5% a vyšší [4]. Jen pro zajímavost, CIFAR-100 dosahuje úspěšnosti pouze okolo 75%.

Pro dosažení větší přesnosti jsou zapotřebí rozsáhlé architektury a často i speciální algoritmy, které přesahují rámec knihovny vytvořené v této práci. U menších architektur, a zejména projektů, se za úspěch považuje již přesnost vyšší než 60%. Nejvyšší dosaženou úspěšností byl výsledek 73.59% na architektuře se třemi konvolučními vrstvami (celkem 12, 24 a 120 filtrů) a dvěma plně propojenými vrstvami (84 a 10 neuronů). Další detaily lze nalézt v kapitole 7.3.3. Získaný výsledek lze pro danou architekturu opět považovat za úspěch (architektury s mnohem vyšším počtem vrstev a filtrů dosahují úspěšnosti pouze o několik málo procent vyšší [6]).

7.2.3 Další datasety

Zatímco dva výše uvedené datasety lze nalézt prakticky v každé práci, která si dává za cíl vyhodnotit určitou novou architekturu nebo nový nápad, tak existují i další datasety, které se však používají méně. Při validaci této knihovny nebyly použity, proto zde budou jen stručně zmíněny.



Obrázek 7.5: Ukázka CIFAR-10 datasetu [22]

STL-10 *STL-10* dataset [8] je podobný CIFAR-10, avšak velikost všech obrázků je 96×96 pixelů. Úspěšnost se pohybuje okolo 75% [4]. Je určen spíše pro učení bez učitele, což není případ této práce.

SVHN *SVHN* [30] je zkratka z anglického *Street View House Numbers*. Jedná se tedy o sadu popisných čísel sesbíranou při vytváření *Google Street View* služby. Obsahuje více než 600 000 obrázků, na kterých se vyskytuje jedna a více číslic. Kolem jednotlivých číslic jsou vykresleny obdélníky, aby bylo usnadněno využití. Existuje ale také odvozený dataset s předzpracováním, kde velikost je normalizována na 32×32 pixelů a vzorek je vycentrován na jednu číslici. Ty nejlepší klasifikátory zde dosahují úspěšnosti až 98.3% a více [4].

7.3 Experimenty s nezávislostí na datovém typu

V této kapitole bude vyhodnocena nejen úspěšnost implementace nezávislosti knihovny na datovém typu a implementace datového typu `FixedPoint`, ale i vliv změny datového typu na schopnost sítě provádět inferenci a učit se.

Během počátečních experimentů bylo zjištěno, že trénování v této reprezentaci je velmi problematické – zejména pro menší bitové šířky. V následujících odstavcích budou tato zjištění popsána a na základě toho budou navrženy cílové experimenty.

Doporučení Jedním z parametrů, na který je třeba si dávat velký pozor, je učící koeficient η. V této reprezentaci může snadno dojít k divergenci, ze které se pak síť těžko zotavuje.

To platí zejména pro nižší bitové šířky, kde častěji dochází k přetékání. Čím menší bitová šířka, tím menší učící koeficient musí být, aby bylo zaručeno, že síť bude stále konvergovat.

V případě, že síť chceme převádět do aproximovaného datového typu se šířkou větší než 16 bitů (a dostatečným počtem bitů v celé i desetinné části), tak příliš nezáleží na zvolených aktivačních funkcích, natrénovaných vahách atd. V případě menších bitových šířek je to ale velmi podstatné. Na základě experimentů bylo zjištěno, že nejvhodnější je použití aktivační funkce *Tanh* ve skrytých vrstvách a případně *Leaky ReLU* ve vrstvě výstupní (což pak vyžaduje použití ztrátové funkce *střední kvadratické chyby* i pro klasifikaci). Je také vhodné provádět trénování s nižším koeficientem učení a případně i s regularizací ve formě *weight decay*.

Při učení konvolučních neuronových sítí, zejména pro datový typ `FixedPoint`, se může stát, že trvá dlouhou dobu, než síť začne konvergovat. Proto je vhodné využít optimalizátoru *Adam*, který zajistí rychlejší konvergenci na počátku, avšak sám upravuje učící koeficient takovým způsobem, že se lépe vyhne divergenci v pozdějších fázích učení.

Experimentálně bylo také zjištěno, že při dotrénování sítě převedené z typu s plovoucí řádovou čárkou na typ s pevnou řádovou čárkou, je často vhodné použití jednoduchého optimalizátoru stochastického gradientního sestupu bez momenta.

Tato doporučení však silně závisí na zvolené architektuře, úloze a dalších parametrech. Je tedy vhodné sledovat průběh trénování a případně parametry nebo architekturu sítě upravit.

Návrh experimentů Experimenty byly vždy provedeny pětkrát nad danou architekturou, vždy s jiným počátečním nastavením generátoru náhodných čísel. Výsledky jsou pak průměrem z těchto pěti běhů.

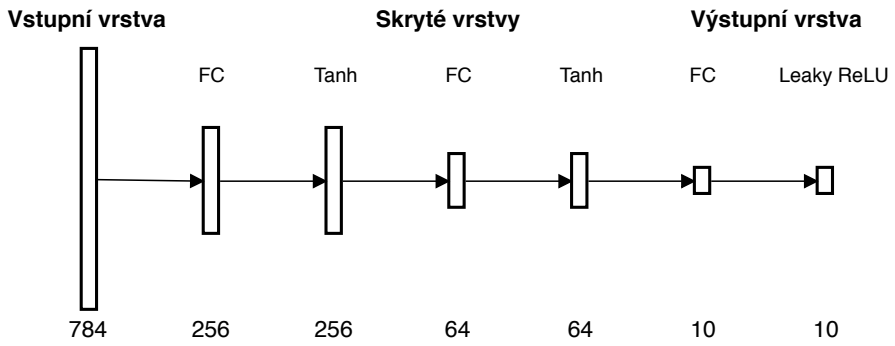
Vzhledem k závislosti na koeficientu učení byla vždy zvolena základní hodnota tohoto parametru a v případě, že došlo k divergenci (pro menší bitové šířky), tak byl koeficient snížen a experiment opakován. V rámci každého běhu tedy jeden konkrétní experiment s jedním konkrétním datovým typem mohl být spuštěn s více koeficienty učení – z výsledků pak vždy byl vybrán ten nejlepší jakožto výsledek celého experimentu pro daný datový typ.

Součástí odevzdání jsou soubory s architekturami sítí, `shell` skripty používané ke spouštění experimentů a také textový záznam průběhu jednotlivých experimentů (pro poslední z pěti běhů).

7.3.1 Běžná neuronová síť

Několik plně propojených vrstev za sebou v zásadě odpovídá neuronové síti. Tento typ sítí byl zvolen pro první experiment vzhledem k jednoduchosti a očekávané lepší responzivitě na snižování bitové šířky. Zjištěné poznatky pak budou aplikovány na experimenty s konvolučními neuronovými sítěmi.

Zvolena byla třívrstvá neuronová síť s počtem vstupních neuronů 784, počtem neuronů ve skrytých vrstvách 256 a 64. Ve výstupní vrstvě je pak 10 neuronů. Skryté vrstvy používají aktivační funkci *Tanh* a výstupní vrstva *Leaky ReLU*. Architekturu lze vidět na obrázku 7.6. Experiment byl prováděn na datové sadě *MNIST*.



Obrázek 7.6: Architektura pro experimenty s neuronovou sítí a datovým typem `FixedPoint`, *FC* značí plně propojenou vrstvu, *Tanh* a *Leaky ReLU* příslušné aktivační vrstvy, čísla označují počet neuronů ve vrstvě

Experiment se skládal ze dvou částí. V první části byla neuronová síť během 10 epoch natrénována na datovém typu `float`, poté proběhlo převedení a dotrénování s různými datovými typy pro váhy a inferenci po dobu dalších 5 epoch.

V tabulce 7.1 je shrnuta úspěšnost ihned po převodu a průměrná nejlepší dosažená úspěšnost během pěti epoch dotrénování. Uvedený datový typ byl totožný pro `ForwardType` a `WeightType`. `BackwardType` byl vždy nastaven na typ `float`, s výjimkou experimentu s typem `double`, kdy i `BackwardType` byl nastaven na tento datový typ (jinak by totiž experiment neměl smysl).

Datový typ	**Úspěšnost na validační sadě [%]**	
	Před dotrénováním	**Po dotrénování**
double	97.37	97.93
float	97.37	97.92
FixedPoint$<16,16>$	97.36	97.92
FixedPoint$<12,4>$	11.39	96.00
FixedPoint$<8,8>$	96.77	97.91
FixedPoint$<4,12>$	97.20	97.90
FixedPoint$<6,2>$	14.69	85.84
FixedPoint$<4,4>$	9.01	95.94
FixedPoint$<2,6>$	23.75	78.20

Tabulka 7.1: Vliv reprezentace na klasifikační přesnost neuronové sítě, trénování provedeno na `float` po dobu 10 epoch, dotrénování poté na daném typu po dobu 5 epoch

V tabulce 7.1 si lze povšimnout, že počet bitů za řádovou čárkou má vliv především na úspěšnost ihned po převedení reprezentace. Dále platí, že pokud je počet bitů před i za řádovou čárkou dostatečný (již zmiňovaných šestnáct a více bitů), tak vliv aproximace je velmi malý nebo dokonce žádný.

Ve druhém experimentu je pak stejná neuronová síť od počátku trénována na daném typu (kdy typ `BackwardType` je stále `float`, opět s výjimkou experimentu s datovým typem `double`). Trénování trvalo 10 epoch a výsledek v tabulce 7.2 je pak průměrem z nejlepších dosažených.

Datový typ	Úspěšnost na validační sadě [%]
double	97.35
float	97.35
FixedPoint$< 16, 16 >$	97.38
FixedPoint$< 12, 4 >$	96.20
FixedPoint$< 8, 8 >$	97.36
FixedPoint$< 4, 12 >$	97.27
FixedPoint$< 6, 2 >$	86.34
FixedPoint$< 4, 4 >$	95.99
FixedPoint$< 2, 6 >$	86.65

Tabulka 7.2: Výsledky trénování neuronové sítě po dobu 10 epoch od počátku v daném typu

Jak je patrné z tabulky 7.2, tak vliv bitové šířky je velmi podobný jako v předchozím experimentu. V případě neuronových sítí tedy příliš nezáleží na tom, zda je síť nejprve trénována na datovém typu s plovoucí řádovou čárkou a poté převedena anebo ihned trénována na tomto datovém typu. Pro 16 a více bitů je vliv na úspěšnost velmi malý až žádný, v případě 8 bitů je již vliv znatelný, ale pro jednodušší úlohy by bylo možno použít i tuto reprezentaci.

Na základě výsledků experimentů s obyčejnými neuronovými sítěmi byly pro další zkoumání vlivu aproximace na konvoluční neuronové sítě vybrány datové typy `double`, `float`, `FixedPoint64<16,16>`, `FixedPoint<8,8>` a `FixedPoint<4,4>`.

7.3.2 Konvoluční neuronová síť

U konvolučních neuronových sítí se v případě aproximace jedná o daleko větší problém a to zejména pro datové typy s nízkou bitovou šířkou. To bude ukázáno v následujících experimentech, které byly prováděny nad architekturou založenou na *LeNet-5*. Specifikaci lze nalézt v souboru `results/cnn_experiments/net.xml` – síť obsahuje tři konvoluční vrstvy (s 6, 16 a 120 filtry) a dvě plně propojené vrstvy (s 84 a 10 neurony).

V prvním experimentu byla konvoluční neuronová síť trénována po dobu 10 epoch na datovém typu `float`, poté byly datové typy inference a vah převedeny do daného datového typu (opět se změnou `BackwardType` pro `double`) a po dobu 5 epoch proběhlo dotrénování.

Datový typ	Úspěšnost na validační sadě [%]	
	Před dotrénováním	Po dotrénování
double	98.60	99.17
float	98.60	99.17
FixedPoint$< 16, 16 >$	86.37	99.15
FixedPoint$< 8, 8 >$	86.91	99.13
FixedPoint$< 4, 4 >$	10.58	79.59

Tabulka 7.3: Vliv reprezentace na klasifikační přesnost konvoluční neuronové sítě, trénování provedeno na typu `float` po dobu 10 epoch, dotrénování poté na daném typu po dobu 5 epoch

V tabulce 7.3 si lze povšimnout, že konvoluční neuronová síť je schopna pracovat bez problému v 16 a více bitech, avšak 8 bitů nestačí. Lze si také povšimnout, že úspěšnost ihned po převedení klesla mnohem razantněji, než v případě obyčejných neuronových sítí.

V případě druhého experimentu byl tedy typ `ForwardType` nastaven vždy na datový typ `FixedPoint<8,8>` (který měl uspokojující výsledky) a měnil se pouze datový typ `WeightType`.

Datový typ vah	**Úspěšnost na validační sadě** [%]	
	Před dotrénováním	**Po dotrénování**
FixedPoint$< 8, 8 >$	86.91	99.13
FixedPoint$< 4, 4 >$	81.97	98.97
FixedPoint$< 1, 3 >$	31.85	98.61
FixedPoint$< 2, 2 >$	10.05	97.67
FixedPoint$< 1, 1 >$	9.80	93.77

Tabulka 7.4: Vliv reprezentace na klasifikační přesnost CNN, trénování provedeno na typu `float` po dobu 10 epoch, dotrénování poté s `FixedPoint<8,8>` pro inferenci a daným datovým typem pro váhy

V tabulce 7.4 si lze všimnout, že pokud je výpočet dostatečně přesný, tak stačí i čtyři bity pro reprezentaci vah a schopnost sítě učit se je taktéž prakticky neovlivněna. Pro jednoduché úlohy by pak šlo použít i reprezentaci s pouhými dvěma bity (ve které lze vyjádřit jen 4 hodnoty: $-1.0, -0.5, 0.0$ a 0.5).

V rámci posledního experimentu byla konvoluční neuronová síť od počátku trénována ve zvoleném datovém typu po dobu 5 epoch. V případě typu `double` je i typ zpětného průchodu nastaven na `double`, v jiných případech je tento typ vždy `float` a typ pro váhy a dopředný průchod je pak zapsán v tabulce 7.5.

Datový typ	**Úspěšnost na validační sadě** [%]
double	98.57
float	98.62
FixedPoint$< 16, 16 >$	98.79
FixedPoint$< 8, 8 >$	98.67
FixedPoint$< 4, 4 >$	64.38

Tabulka 7.5: Výsledky trénování konvoluční neuronové sítě od počátku na daném datovém typu po dobu 10 epoch

V tabulce 7.5 si lze povšimnout, že při trénování sítě od počátku se výsledky příliš neliší pro 16 bitů a více. Na 8 bitech je síť schopna učení, avšak pokles přesnosti sítě při klasifikaci je značný. Opět ale platí, že je možno síť trénovat od počátku v dostatečně „širokém“ aproximovaném datovém typu bez ztráty přesnosti.

Zarážející může být, že trénování od počátku prováděné v datovém typu `FixedPoint` má o několik setin procenta vyšší úspěšnost než je tomu v případě datového typu s plovoucí řádovou čárkou. Stejně tomu bylo i u obyčejných neuronových sítí. Je ale třeba poznamenat, že se jedná pouze o jednotky lépe klasifikovaných obrázků.

7.3.3 Aproximace nejlepších dosažených výsledků

V kapitole 7.2 byly zmíněny nejlepší dosažené výsledky pro datové sady *MNIST* a *CIFAR-10*. Těch bylo dosaženo tak, že byla natrénována zmíněná architektura na datovém typu `float` (pro všechny tři typové aliasy).

Takto natrénovaná síť pak byla dále dvakrát dotrénována se stejnými nastaveními učení po dobu dalších pěti epoch – jednou opět na datovém typu `float` a podruhé na datovém typu `FixedPoint<8,8>` pro váhy a inferenci (zpětný průchod byl stále prováděn v datovém typu s plovoucí řádovou čárkou).

Datový typ `FixedPoint<8,8>` byl zvolen proto, že se jedná o nejmenší bitovou šířku, pro kterou pokles úspěšnosti v předcházejících experimentech nebyl velký.

Datová sada	**Úspěšnost na validační sadě** [%]	
	`float`	`FixedPoint<8,8>`
MNIST	99.37	99.37
CIFAR-10	73.59	73.26

Tabulka 7.6: Výsledky dotrénování nejlepších sítí na daných datových sadách a při daném typu

Výsledky lze vidět v tabulce 7.6. V případě datové sady *MNIST* má síť převedená do aproximované reprezentace stejnou úspěšnost. V případě datové sady *CIFAR-10* je úspěšnost mírně nižší – pokud bychom ale síť trénovali jen o jednu epochu déle, tak by úspěšnost aproximované reprezentace byla již 73.54%, což se dá považovat za dostatečně blízké původní hodnotě.

Jak lze tedy vidět, tak i konvoluční neuronové sítě, které byly trénovány velmi dlouhou dobu za účelem dosažení co nejlepších možných výsledků, lze převést do aproximované reprezentace v podobě pevné řádové čárky – a to téměř bez ztráty přesnosti.

Nejlepší dosažené výsledky, jak v případě datového typu s plovoucí řádovou čárkou, tak v případě typu s pevnou řádovou čárkou, jsou součástí odevzdání.

7.3.4 Vyhodnocení

Z provedených experimentů je patrné, že implementace datového typu s pevnou řádovou čárkou byla úspěšná, stejně jako implementace datové nezávislosti. Byly měněny všechny tři typové aliasy a síť byla schopná učení a inference. Je tedy možné nad knihovnou vytvořit navazující práce, které budou dále zkoumat vliv aproximovaných reprezentací či funkcí.

Co se týče vlivu použití reprezentace s pevnou řádovou čárkou na přesnost sítě, tak očekáváným výsledkem bylo, že datový typ s plovoucí řádovou čárkou bude nejpřesnější. Dále v případě typu `FixedPoint` bylo předpokladem, že se snižujícím se počtem bitů bude klesat úspěšnost na validační sadě. Tyto předpoklady se potvrdily a lze říci, že neuronové sítě a konvoluční neuronové sítě lze provádět v této reprezentaci – avšak závisí na vhodné volbě počtu bitů vzhledem k řešené úloze.

Bylo také zmíněno, že trénování sítě v této reprezentaci je problematické, zejména pro nižší bitové šířky. Je tedy vhodné, aby případný uživatel vyzkoušel různá nastavení a sledoval průběh trénování a případně parametry upravil za účelem získání co nejlepších výsledků. K tomu může využít doporučení uvedená v předcházejících odstavcích. Velmi závisí na architektuře sítě, hyperparametrech, úloze a dalších nastaveních. Získané výsledky se tedy mohou lišit – a to jak v pozitivním, tak negativním smyslu.

Kapitola 8

Závěr

V této diplomové práci byla nastíněna problematika neuronových a konvolučních neuronových sítí. Byly identifikovány jejich podstatné části, které by měla každá knihovna implementovat, a jejich problémy, které je potřeba vyřešit.

Na základě těchto znalostí poté byla navržena a v jazyce `C++` implementována knihovna pro práci s konvolučními neuronovými sítěmi. A to včetně rozšíření souvisejícím s nezávislostí na datovém typu, především se zaměřením na datový typ s pevnou řádovou čárkou. Knihovna, pojmenovaná *TypeCNN* s odkazem na nezávislost na datovém typu, je volně dostupná pod licencí *MIT* na adrese `http://github.com/rekpet/TypeCNN`.

Implementovaná knihovna byla vyhodnocena na charakteristických úlohách *MNIST* a *CIFAR-10*, kdy bylo dosaženo uspokojujících hodnot. Knihovna byla taktéž porovnána s dalšími knihovnami volně dostupnými na internetu. Z pohledu kvality učení se vyrovná i knihovnám běžně používaným v praxi. Z pohledu časové náročnosti zase dokáže překonat knihovny podobného rozsahu. Je tedy zřejmé, že knihovna je plně funkční, avšak do budoucna ji lze dále vylepšovat – jak s ohledem na množství funkcionality, tak zejména s ohledem na rychlost. Další zefektivňování knihovny totiž bylo omezeno vzhledem k časové náročnosti prováděných experimentů – zejména experimentů zkoumajících vliv aproximované reprezentace.

Rozšířením pak byla implementace datové nezávislosti knihovny, Z provedených experimentů je patrné, že tento bod byl také splněn. V rámci případové studie byl ověřován vliv aproximace převedením do reprezentace s pevnou řádovou čárkou na (konvoluční) neuronové sítě. Bylo zjištěno, že použitá reprezentace dokáže vhodně aproximovat provádění konvoluční neuronové sítě na datovém typu s pevnou řádovou čárkou místo typu s plovoucí řádovou čárkou. Záleží však na velikosti sítě a bitové šířce zvoleného typu. Stejně tak bylo ukázáno, že i když inference tuto šířku požaduje vyšší, tak v případě vah je možno bitovou šířku omezit více.

Vzhledem k tomuto rozšíření se tak knihovna nabízí k použití pro další výzkumnou činnost, která může dále prozkoumat vliv reprezentace s pevnou řádovou čárkou na konvoluční neuronové sítě, ale i dalších reprezentací nebo třeba aproximovaných funkcí.

Literatura

[1] Abadi, M.; Agarwal, A.; Barham, P.; aj.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015.
URL https://www.tensorflow.org/

[2] Abadi, M.; Agarwal, A.; Barham, P.; aj.: Fixed Point Quantization. *TensorFlow dokumentace*, 2018.
URL https://www.tensorflow.org/performance/quantization

[3] Ali, K.: Implementing Fixed-Point Numbers in C++. *Khuram Ali | Code is poetry*, 2013.
URL https://alikhuram.wordpress.com/2013/05/20/implementing-fixed-point-numbers-in-c/

[4] Benenson, R.: What is the class of this image? 2016.
URL http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

[5] Britz, D.: Understanding Convolutional Neural Networks for NLP. *WILDML (Artifical Intelligence, Deep Learning and NLP)*, 2015.
URL http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

[6] Chollet, F.; aj.: Keras: Example of architecture for CIFAR-10 dataset.
URL
https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py

[7] Chollet, F.; aj.: Keras. 2015.
URL https://keras.io

[8] Coates, A.; Ng, A.; Lee, H.: An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, *Proceedings of Machine Learning Research*, ročník 15, Fort Lauderdale, FL, USA: PMLR, 2011, s. 215–223.
URL http://proceedings.mlr.press/v15/coates11a/coates11a.pdf

[9] Du, S. S.; Lee, J. D.; Tian, Y.; aj.: Gradient Descent Learns One-hidden-layer CNN: Don't be Afraid of Spurious Local Minima. 2017.
URL https://arxiv.org/pdf/1712.00779.pdf

[10] Eichner, H.: Neural Net for Handwritten Digit Recognition in JavaScript. 2014.
URL http://myselph.de/neuralNet.html

[11] *can1357*: SimpleCNN: Simple Convolutional Neural Network Library. *GitHub*, 2016.
URL https://github.com/can1357/simple_cnn/

[12] Finnerty, A.; Ratigner, H.: Reduce Power and Cost by Converting from Floating Point to Fixed Point. 2017.
URL https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf

[13] Fortuner, B.: Machine Learning Cheatsheet. 2017.
URL http://ml-cheatsheet.readthedocs.io/

[14] Gargano, M. L.; Marose, R. A.; von Kleeck, L.: An application of artificial neural networks and genetic algorithms to personnel selection in the financial industry. In *Proceedings First International Conference on Artificial Intelligence Applications on Wall Street*, 1991, s. 257–262.

[15] Goodfellow, I.; Bengio, Y.; Courville, A.: *Deep Learning*. MIT Press, 2016.
URL http://www.deeplearningbook.org

[16] Gysel, P.; Pimentel, J.; Motamedi, M.; aj.: Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2018, doi:10.1109/TNNLS.2018.2808319.

[17] He, K.; Zhang, X.; Ren, S.; aj.: Deep Residual Learning for Image Recognition. *CoRR*, ročník abs/1512.03385, 2015.
URL http://arxiv.org/abs/1512.03385

[18] Jacobson, L.: Introduction to Artificial Neural Networks Part 2 - Learning. 2014.
URL http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-2-learning/8

[19] Janoušek, V.: Machine Learning and Neural Networks. *Slidy k předmětu SIN (Inteligentní systémy). Fakulta Informačních Technologií, Vysoké Učení Technické v Brně*, 2017.

[20] Jia, Y.; Shelhamer, E.; Donahue, J.; aj.: Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[21] Karpathy, A.: Lecture notes for COMPSCI 682 (Neural Networks: A Modern Introduction). Leden 2015.
URL http://cs231n.github.io/

[22] Krizhevsky, A.; Hinton, G.: Learning multiple layers of features from tiny images. *Diplomová práce, Department of Computer Science, University of Toronto*, 2009.

[23] Krizhevsky, A.; Sutskever, I.; Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, s. 1097–1105.
URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[24] Lecun, Y.; Bottou, L.; Bengio, Y.; aj.: Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998, s. 2278–2324.

[25] LeCun, Y.; Cortes, C.: MNIST handwritten digit database. 2010.
URL http://yann.lecun.com/exdb/mnist/

[26] Matthews, T.: Perceptron: XOR (how & why neurons work together). *Computation neuroscience in Excel*, 2012.
URL http://toritris.weebly.com/perceptron-5-xor-how--why-neurons-work-together.html

[27] McCulloch, W. S.; Pitts, W.: A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, ročník 5, č. 4, 1943: s. 115–133.
URL https://doi.org/10.1007/BF02478259

[28] Minsky, M.; Papert, S.: *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[29] Mrázek, V.; Hrbáček, R.; Vašíček, Z.; aj.: EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Proc. of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, European Design and Automation Association, 2017, ISBN 978-3-9815370-9-3, s. 258–261.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11262

[30] Netzer, Y.; Wang, T.; Coates, A.; aj.: Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.
URL http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf

[31] Nomi, T.: TinyDNN: header only, dependency-free deep learning framework in C++14. *GitHub*, 2013.
URL https://github.com/tiny-dnn/

[32] Reid, S. G.: 10 misconceptions about Neural Networks. *Turing Finance*, 2014.
URL http://www.turingfinance.com/misconceptions-about-neural-networks/

[33] Rosenblatt, F.: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review*, 1958: s. 65–386.

[34] Ruder, S.: An overview of gradient descent optimization algorithms. *CoRR*, ročník abs/1609.04747, 2016.
URL http://arxiv.org/abs/1609.04747

[35] Russakovsky, O.; Deng, J.; Su, H.; aj.: ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, ročník 115, č. 3, 2015: s. 211–252.

[36] So, H.: Introduction to Fixed Point Number Representation. 2006.
URL http://www-inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html

[37] Springenberg, J. T.; Dosovitskiy, A.; Brox, T.; aj.: Striving for Simplicity: The All Convolutional Net. *CoRR*, 2014.
URL http://arxiv.org/abs/1412.6806

[38] Sreeram, J.; Herhut, S.; Kuper, L.: Bringing Parallelism to the Web with River Trail. *RiverTrail*.
URL http://intellabs.github.io/RiverTrail/tutorial/

[39] Strange, A.: Elon Musk's secret fear: Artificial Intelligence will turn deadly in 5 years. *Mashable*, 2014.
URL http://mashable.com/2014/11/17/elon-musk-singularity/

[40] Trivedi, S.; Kondor, R.: Convolutional Neural Networks. *CMSC 35246: Deep Learning*, 2017.
URL http://ttic.uchicago.edu/~shubhendu/Pages/Files/Lecture7_flat.pdf

[41] Werbos, P.: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975.

[42] Zbořil, F.: Neuronové sítě a backpropagation. *Slidy k předmětu SFC (Soft computing), Fakulta informačních technologií, Vysoké učení technické v Brně*, 2016.

Příloha A

Manuál

Způsob použití konzolového i programového rozhraní je popsán v kapitole 6. Ukázkové projekty většinou nepotřebují žádné vstupní parametry a jsou popsány v téže kapitole.

Unix Knihovna byla testována primárně na operačním systému *Ubuntu*, je tedy doporučeno ji využívat na *unixových* systémech. K dispozici je skript `Makefile` a to jak v hlavním adresáři, kde po vyvolání příkazu `make` dojde k vytvoření spustitelného souboru konzolového rozhraní, tak v adresáři `examples/`, kde jsou vytvořeny spustitelné soubory pro jednotlivé ukázkové projekty založené nad knihovnou.

Pro spuštění testů je nutno mít nainstalovaný modul `gtest`[1] a to ve standardním adresáři. Pokud tomu tak není, je nutno upravit soubor `Makefile`. Spustitelný soubor lze pak vytvořit zadáním příkazu `make tests`.

Windows Pro operační systém *Windows* jsou k dispozici soubory pro vývojové prostředí *Visual Studio 2017* ve složce `visualStudio/`. Na tomto operačním systému proběhlo základní testování a nebyly zjištěny žádné problémy. Po otevření lze nalézt projekty jak pro konzolové rozhraní, tak pro ukázkové projekty i testy.

Pro spuštění testů je nutno mít nainstalovaný modul `gtest`. Nastavení projektu testů je v případě programu *Visual Studio* složitější – jednoduchý postup lze nalézt například zde: https://stackoverflow.com/a/47795243/984471.

[1] Dostupné z https://github.com/google/googletest.

Příloha B

Obsah DVD

Přiložené DVD obsahuje tuto zprávu společně se zdrojovými kódy a dále také zdrojové kódy knihovny se všemi dalšími podklady.

Knihovna je taktéž dostupná na adrese http://github.com/rekpet/TypeCNN, avšak neobsahuje tuto diplomovou práci a její zdrojové kódy.

Obsah DVD je detailně popsán v seznamu níže.

- Technická zpráva (`text/`):
  - `sources/` – zdrojové kódy v jazyce LaTeX se skriptem `Makefile` pro přeložení,
  - `DP_xrekpe00.pdf` – tato diplomová práce ve formátu `.pdf`.
- Knihovna (`sources/`):
  - `3rdParty/` – zdrojové soubory knihoven třetích stran použitých v práci,
  - `cli/` – zdrojové kódy konzolového rozhraní,
  - `examples/` – zdrojové kódy ukázkových projektů postavených nad implementovanou knihovnou,
  - `resources/` – vstupní data pro trénování, obsahuje datové sady *MNIST* a *CIFAR-10*,
  - `results/` – obsahuje natrénované konvoluční neuronové sítě a výsledky experimentů,
  - `src/` – zdrojové kódy knihovny pro využití ve vlastním projektu,
  - `tests/` – zdrojové kódy jednotkových testů ve frameworku *Google Test*,
  - `visualStudio/` – soubory pro *Visual Studio 2017* umožňující překlad knihovny na operačním systému *Windows*,
  - `LICENSE` – *MIT* licence knihovny,
  - `Makefile` – skript pro vytvoření spustitelných souborů konzolového rozhraní a testů,
  - `README.md` – soubor s krátkým popisem knihovny.